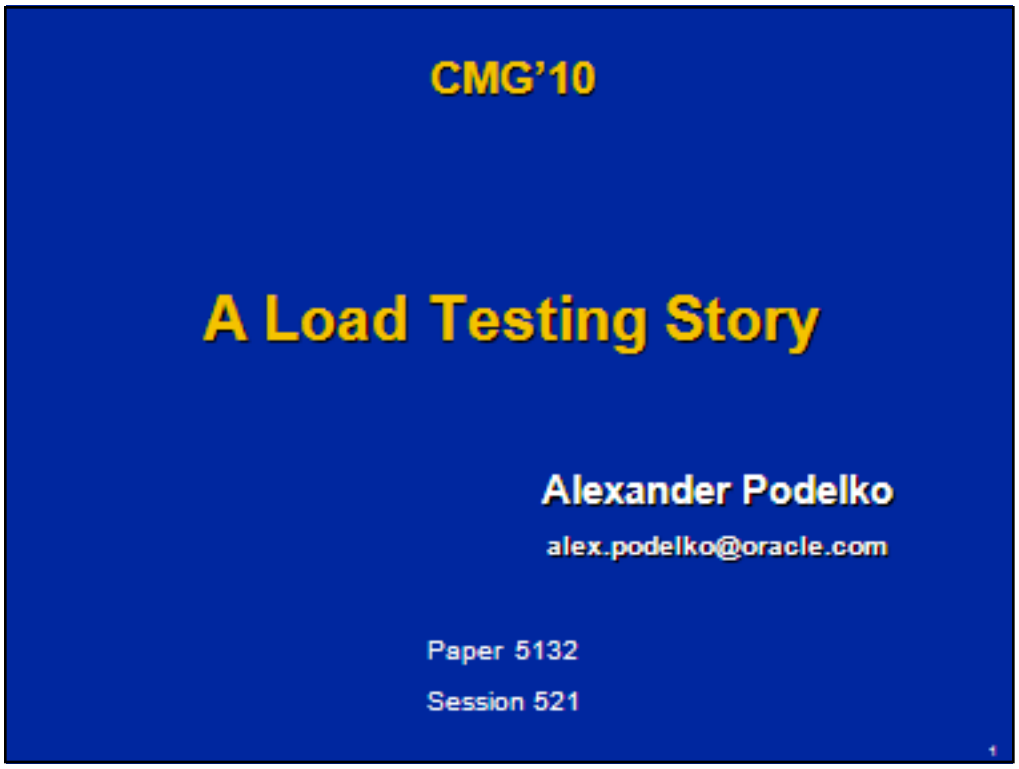
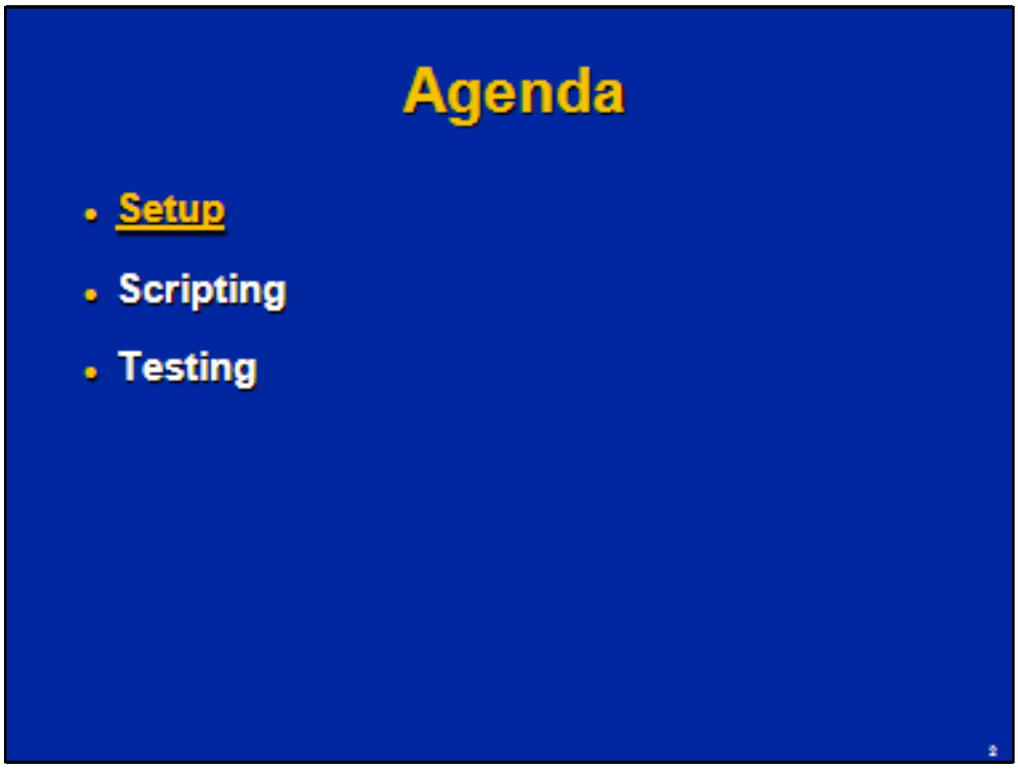
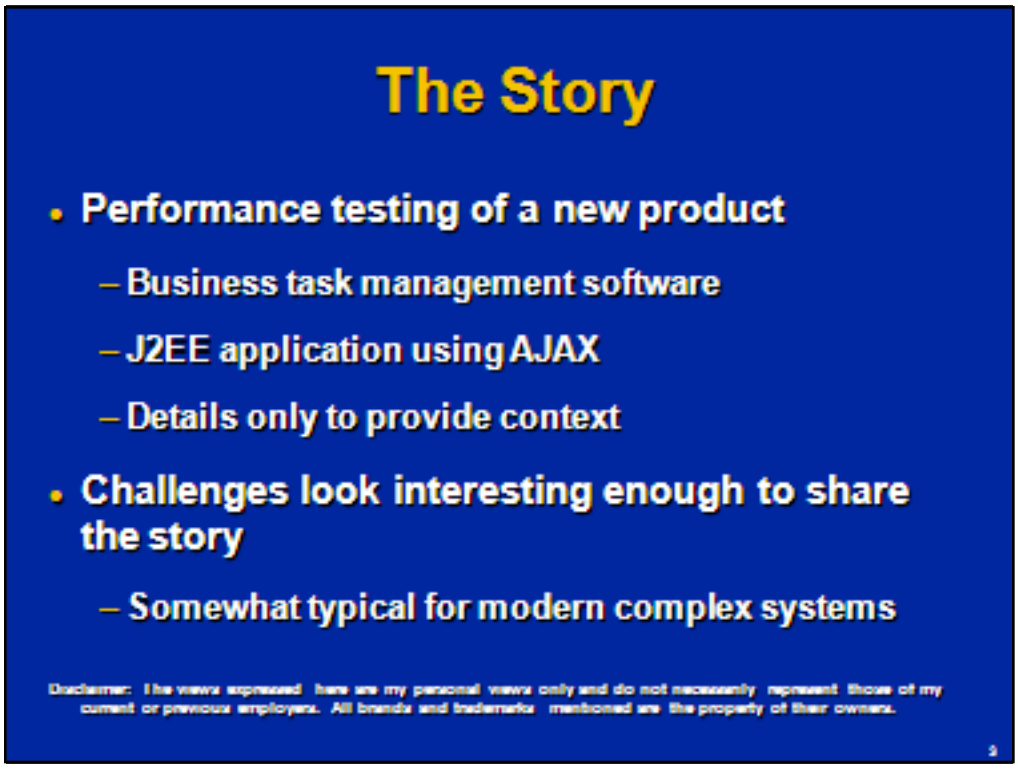


The presentation describes a load testing project chronologically. The scope of the project was to test task management software for performance. It was a new, multi-tier Java application using AJAX technologies and working in close collaboration with other products. The project posed multiple challenges far beyond simple record-and-playback. While details are specific for the project, the challenges and solutions are somewhat typical for complex modern systems. The presentation concentrates on load testing methodology including system setup, scripting, test planning, and test execution.



First the product and test setup would be discussed to provide enough context. Then scripting challenges would be discussed, and in the end the tests planned and executed.

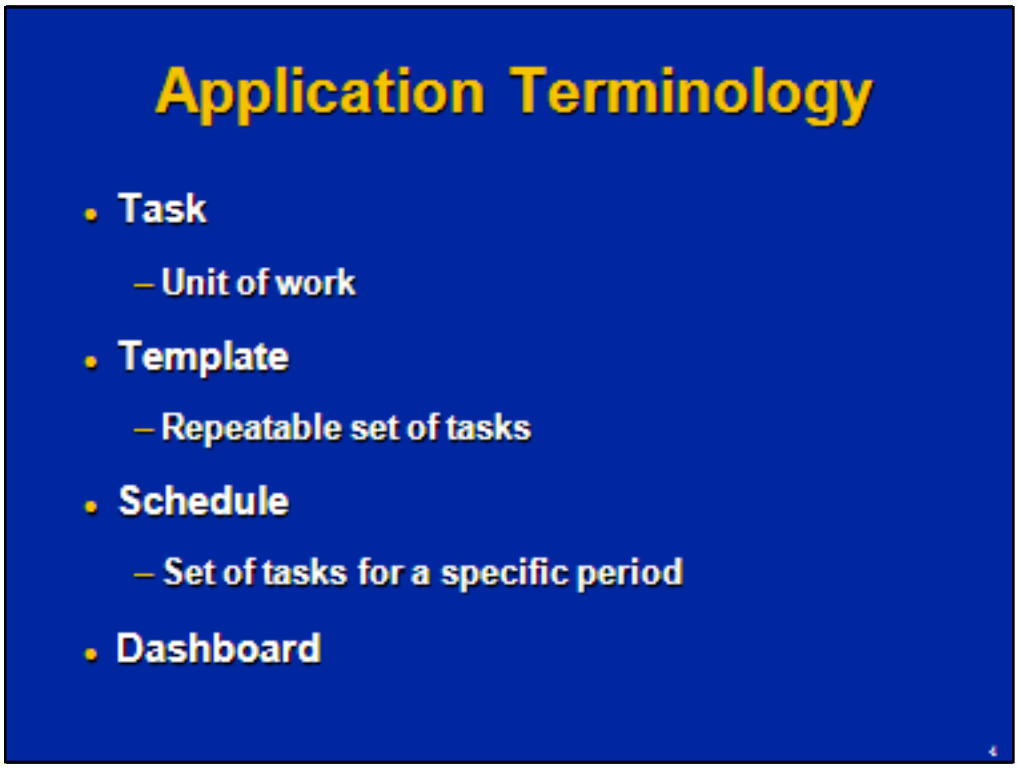
While multiple performance-related issues were found during the testing (as it is usual for a brand new product), they are not discussed here because they are not particularly relevant to the discussed load testing process.



The presentation describes a load testing project chronologically. The scope of the project was to test business task management software for performance. It was a new, multi-tier Java application using Asynchronous JavaScript and XML (AJAX) technologies and working in close collaboration with other products. The name and specific functionality of the product are not really important to the story. Whatever information about the product is mentioned, it is mentioned only to set up a meaningful context for describing the project.

The project posed multiple challenges far beyond simple record-and-playback. While details are specific for the project, the challenges and solutions are somewhat typical for modern complex systems. The presentation concentrates on load testing methodology including system setup, scripting, test planning, and test execution.

I want to state explicitly that the views expressed here are my personal views only and do not necessarily represent those of my current or previous employers. All brands and trademarks mentioned are the property of their owners.



Tasks

A task is a unit of action, for example, data entry or data processing. Users define the tasks that comprise a process. Users can, for example, read task instructions, submit, reassign, approve, or reject tasks. Tasks may be in different states like pending, open, or closed and are changing states during their life cycle. Tasks maybe manual or automated.

Templates

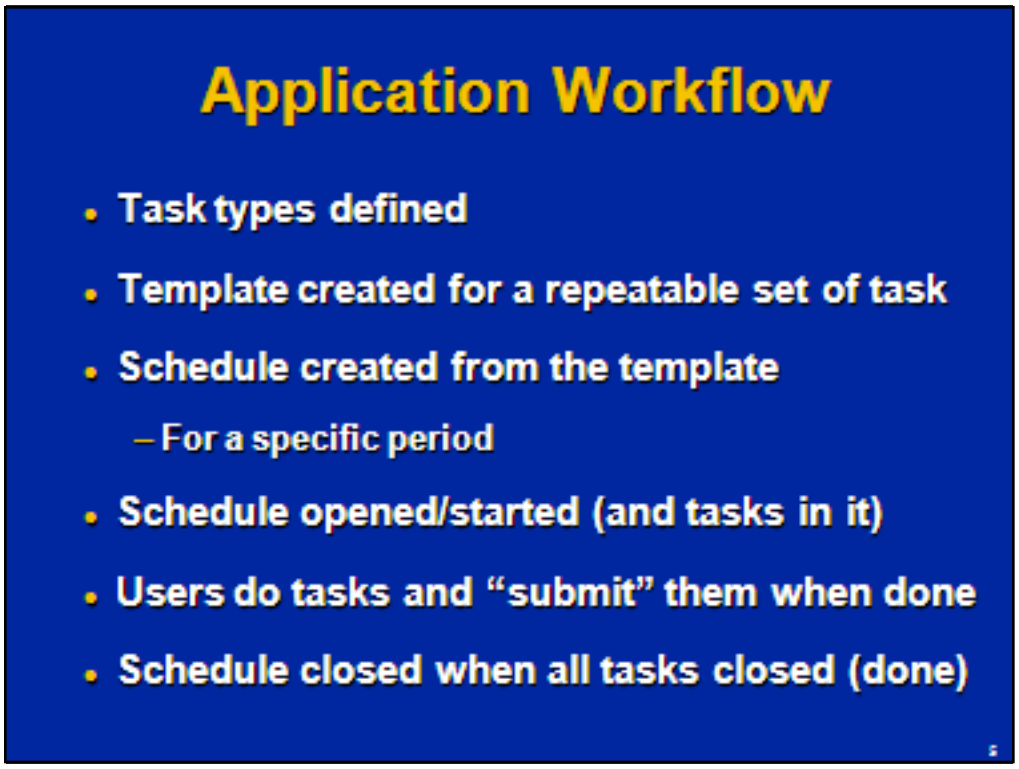
A template is a set of tasks that are repeatable over periods (for example, monthly or quarterly). Tasks inside templates are described relatively to the start time.

Schedules

A schedule defines the chronologically ordered set of tasks that must be executed for a specific period. If created from a template, it translates template's relative days to actual calendar dates. Schedules may be in different states like pending, open, or closed and are changing states during their life cycle.

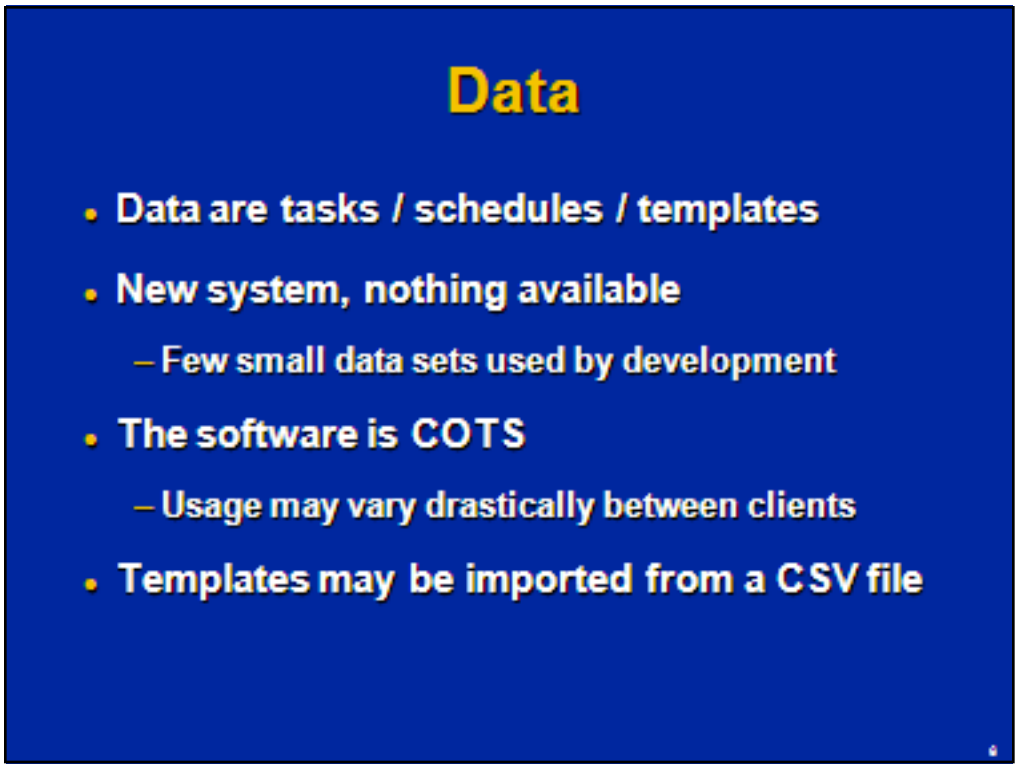
Dashboard

The Dashboard view presents a portal-style interface with views into schedules and high-level summaries into which you can drill down for greater detail.

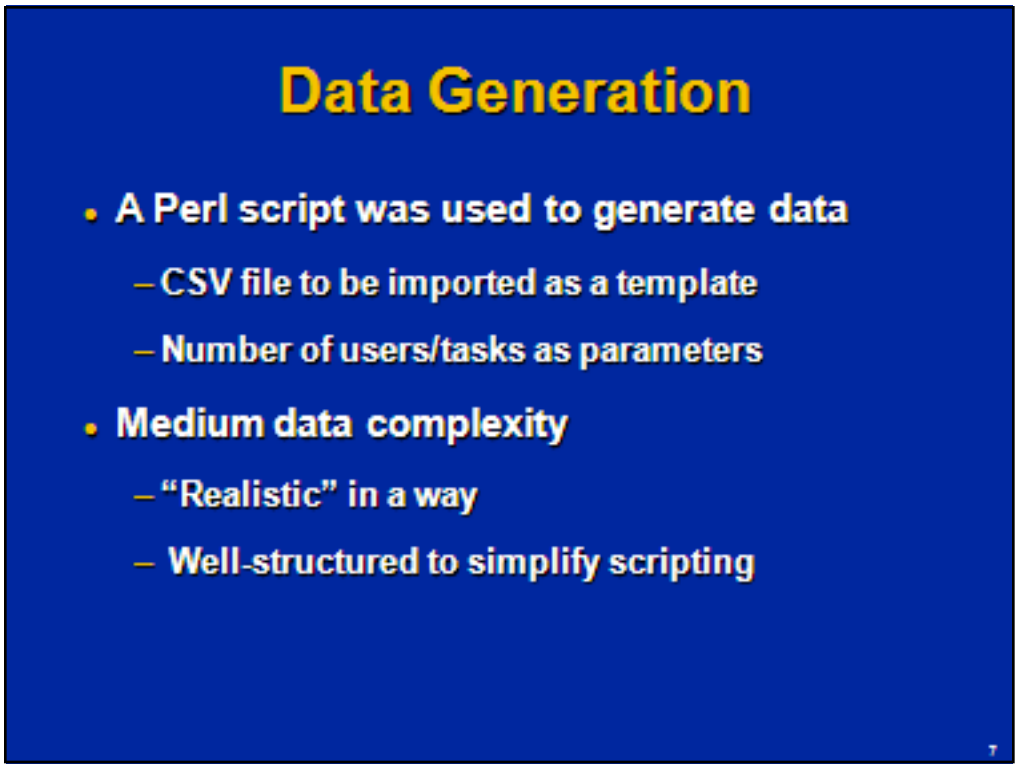


First, task types required for a process are set to ensure consistency across tasks and to leverage predefined product integrations. Then a set of tasks required for a process and repeatable over periods is saved as a template to use for future periods.

After a template is created, it can be used as a source for generating a schedule (a chronological set of tasks), to be run in a period. The generic tasks in the template are applied to actual calendar dates. During the process, users receive email notifications of assigned tasks, and can click links in the email for direct access to assigned tasks. Alternatively, users can log on to review and access assigned tasks in different types of views, for example, the Dashboard, a portal-style interface; or Calendar, Gantt, or Task List views. When users complete tasks, the tasks are sent (submitted) to approvers and can be viewed by other users.



In this particular case data are tasks grouped in schedules and templates. The product is a new "Commercial Off-the-Shelf" (COTS) system, no data were available except small data sets used by developers and functional testers. Usage may vary drastically between clients. Data may be entered through the User Interface (UI) or templates may be imported from a CSV file. The latter way looks preferable for generation of large sets of data.



A Perl script was used to generate data in form of a CSV file to be imported as a template. The number of users, tasks, and task names were parameters (variables set in the beginning of the Perl script).

The data have medium data complexity and are well-structured to simplify scripting. They are "realistic" in a way: the structure is based on speculations on what would be "typical" usage. Every user has 5 tasks assigned and each next task of these 5 is dependent on the previous. So the CSV file looks like:

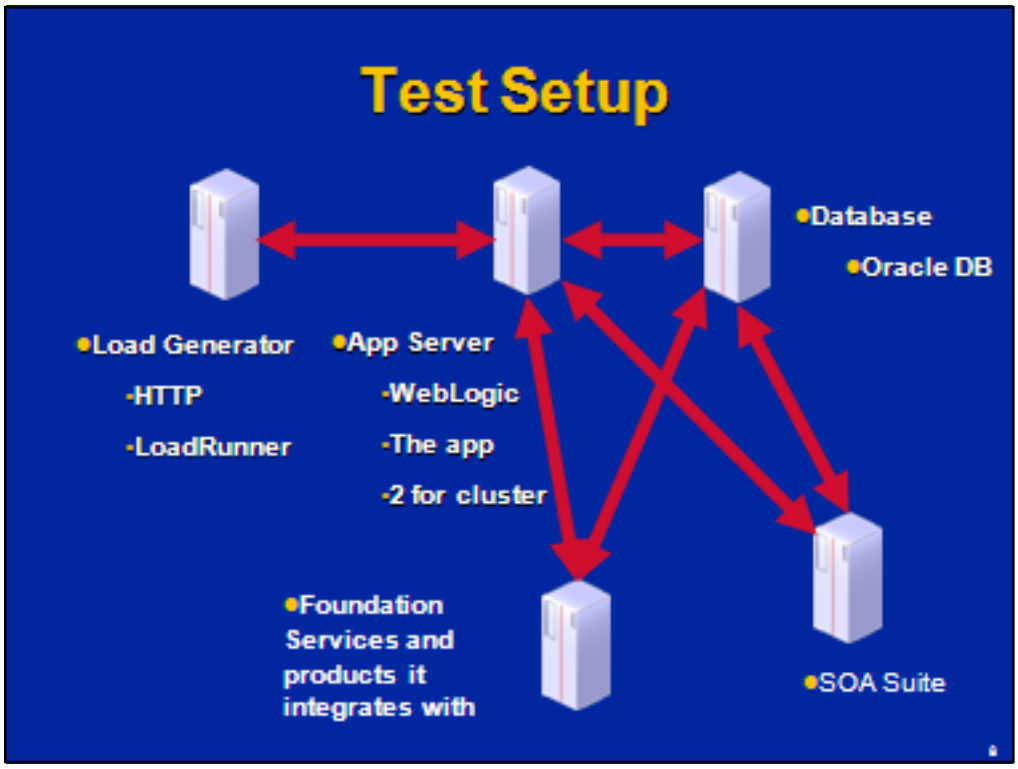
TaskID,Parent,Predecessor1,PredType1,Predecessor2,PredType2,TaskName,Description,Instruction,TaskType,Priority,Owner,Assignee,Approver1,Approver2,StartTime,StartDate,EndTime,EndDate

1,,,,,,task 1,task 1,Read manual,Metadata Load: Classic,Medium,admin,user0001,,,,-12,,-8

2,,,,,,task 2,task 2,Read manual,Metadata Load: Classic,Medium,admin,user0002,,,,-12,,-8

…

21,,1,Finish to Start,,,task 21,task 21,Read manual,Data Load: FDM,Medium,admin,user0001,,,,-7,,-3



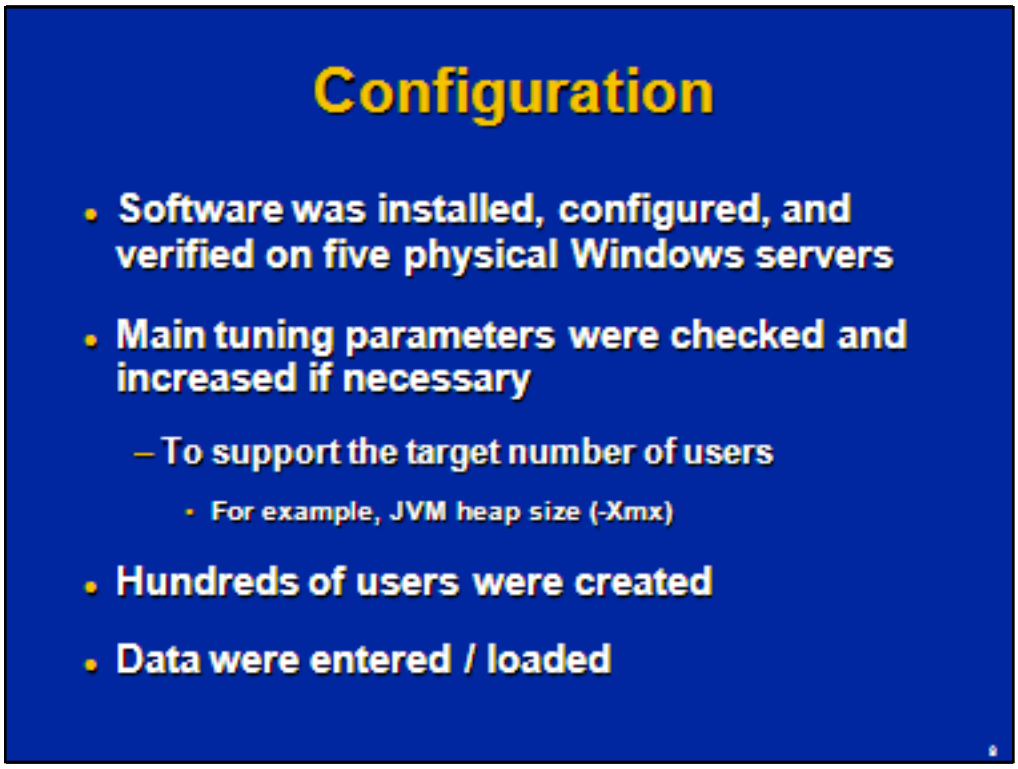
The system under investigation is a multi-tier Java EE (Enterprise Edition) application.

The first tier is a thin client (Browser), replaced by a load generation tool. HP LoadRunner was used in this project to generate load (any sophisticated enough load generation tool maybe used).

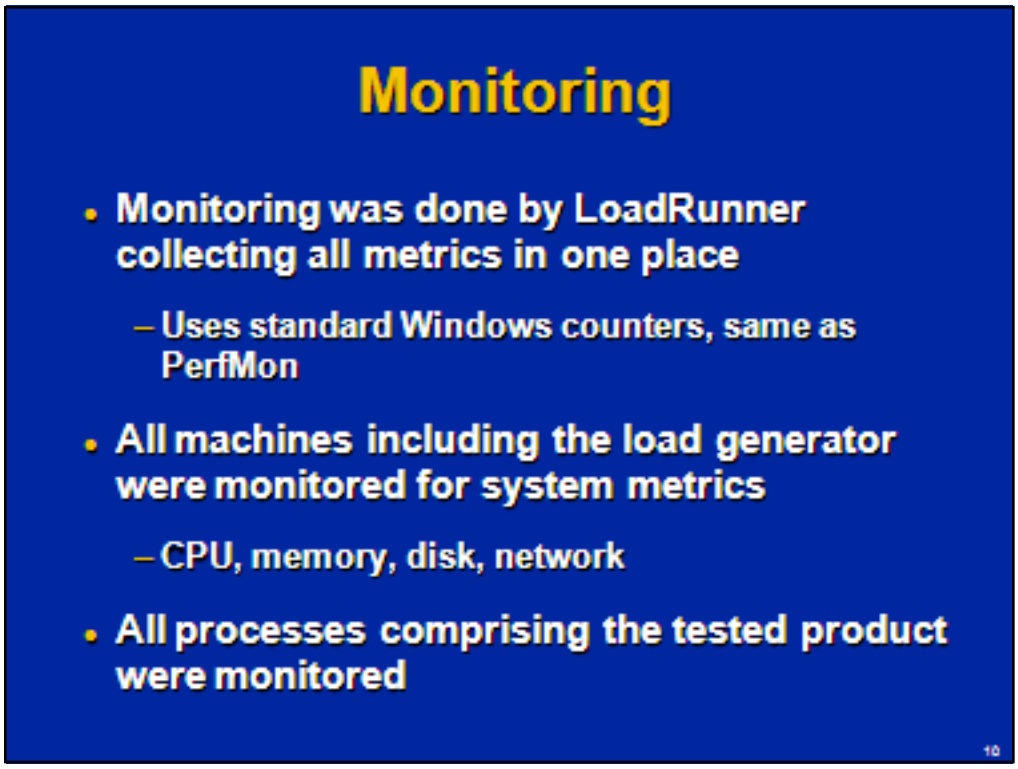
The second tier is the Java EE application itself deployed on WebLogic Application Server. Most tests were done with one application server, but when cluster is explicitly mentioned, two identical servers were used as application servers and load was balanced between them.

The third tier is the database tier. Oracle Database (DB) was used in the project.

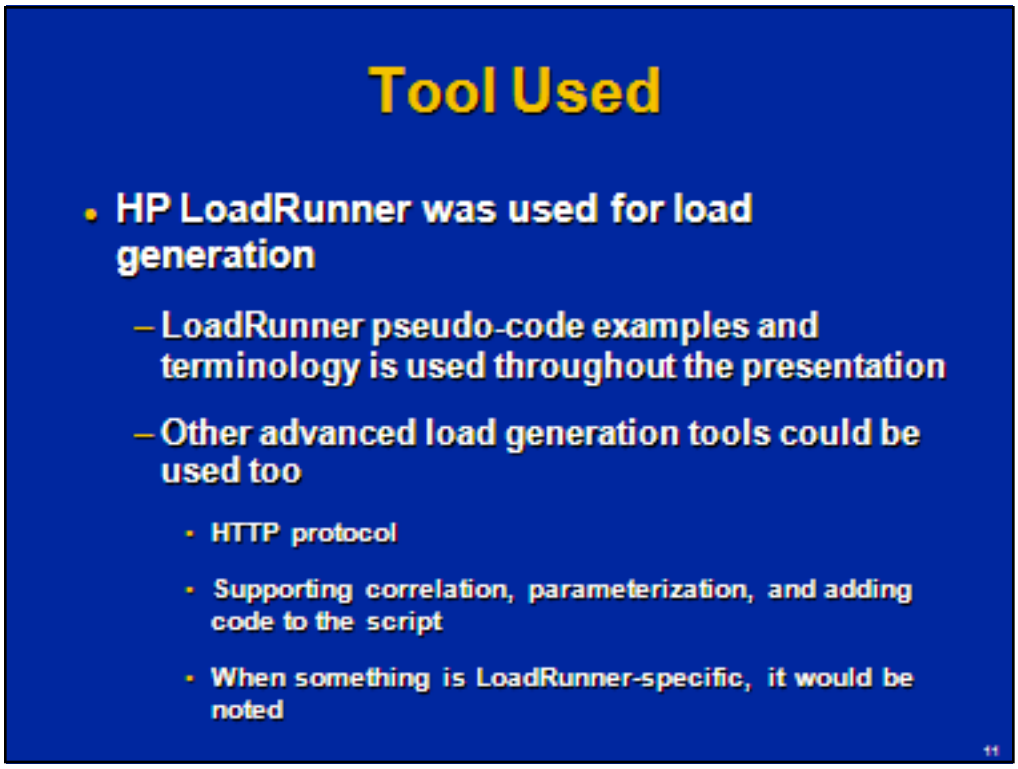
The product is tightly integrated with other products: Foundation Services, Service-Oriented Architecture (SOA) Suite, and many other products it integrates with (to execute tasks).



Software was installed, configured, and verified on five physical four-way servers. Main tuning parameters were checked and increased if necessary to support the target number of users (for example, JVM heap size -Xmx). Hundreds of users were created in the system and got provisioned with necessary access rights. Data were entered (like task types and periods) and loaded from the generated CSV files.



Monitoring was done by LoadRunner collecting all metrics in one place. LoadRunner uses standard Windows counters, same as PerfMon. All machines including LoadGenerator were monitored for system metrics: CPU, memory, I/O, disk. Also all processes comprising the tested product were monitored too, mainly for CPU and memory counters.

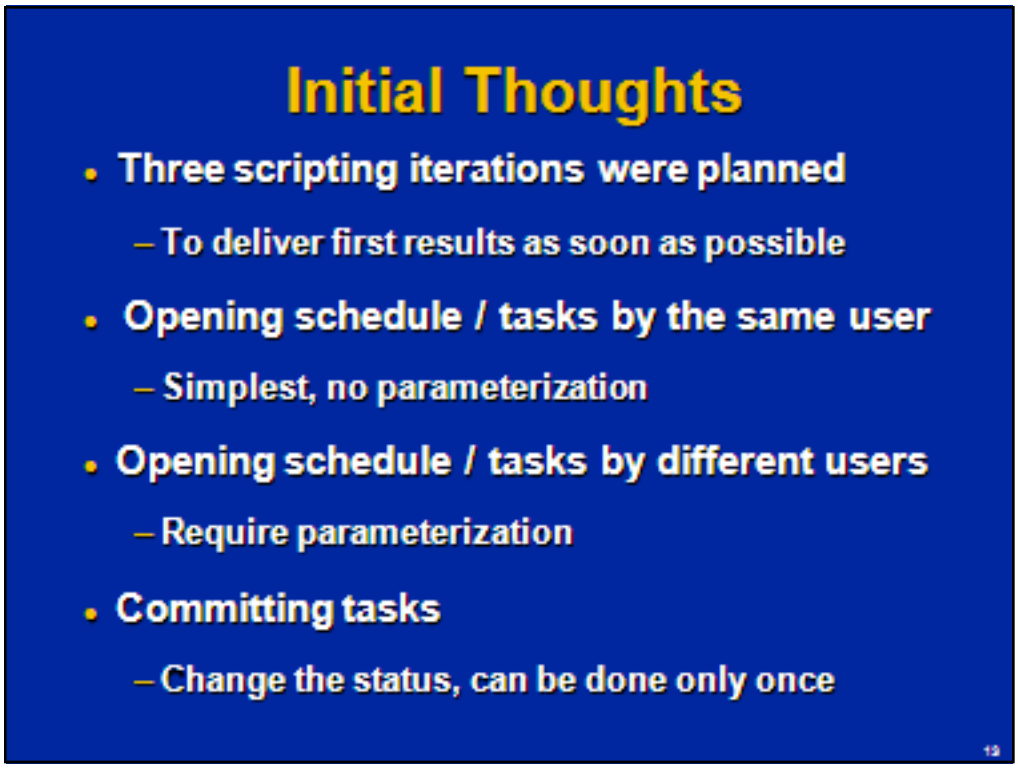


HP LoadRunner was used for load generation, so LoadRunner pseudo-code examples and terminology is used throughout the presentation. The application uses the HTTP protocol to communicate between client (Browser) and server. Other advanced load generation tools could be used too (supporting correlation, parameterization, and adding code to the script), although syntax / terminology would be different. When something is LoadRunner-specific, it would be noted.

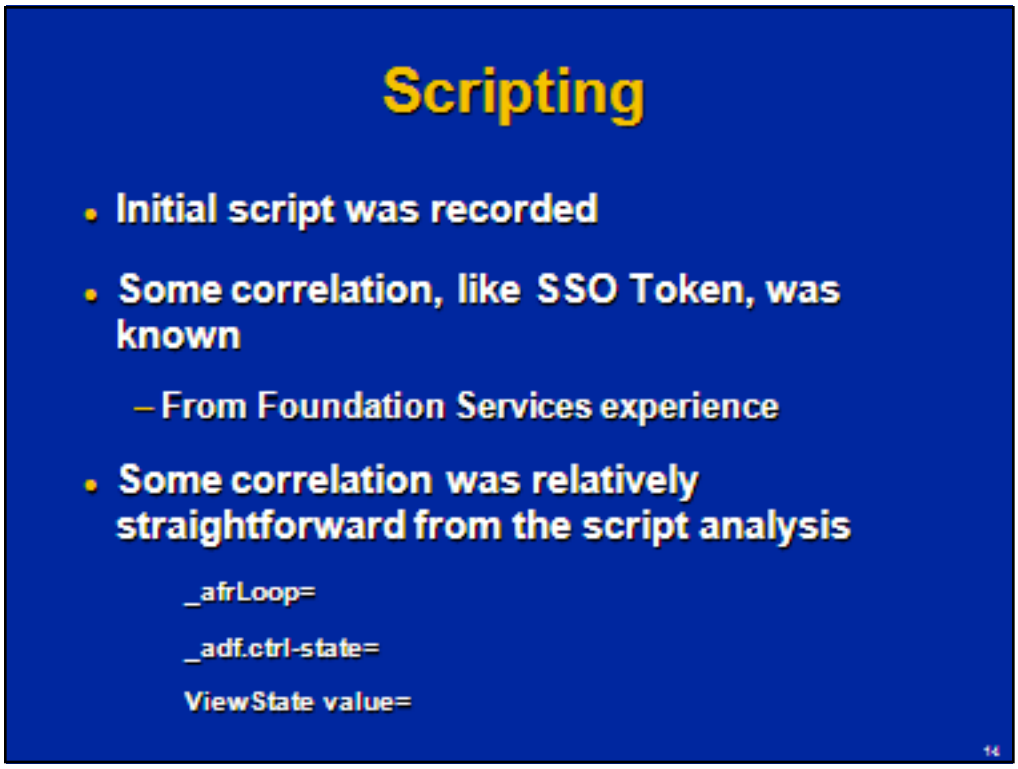
Agenda

- Setup
- Scripting
- Testing

12



As far as it is a brand new product, the first thing would be to verify if recording / playback approach could be used and how difficult would be scripting. Three scripting iteration were planned. The reason was to deliver first results as soon as possible. In the beginning, we create a script for opening schedule / tasks by the same user. This is the simplest case, no change in the system state, no parameterization. When this script would be done and verified, we can parameterize it to open schedule / tasks by different users. When this will be done and verified, we can do special cases like committing tasks, which change the status, so can be done only once.



The traditional approach to load generation was used. The load testing tool recorded the HTTP communication between client and server. The recorded script was then correlated (some values in the script were replaced by values received from the server during playback) and parameterized (some values were replaced by parameters to make sure, for example, that different user names and tasks are used). The load testing tool generates workload creating multiple virtual users (sessions) and playing back the script.

A initial script was recorded. Some correlation, like SSO Token, belongs to Foundation Services, so was known from previous experience. Some other correlation was relatively straightforward from the script analysis, for items in the script like:

_afrLoop=

_adf.ctrl-state=

ViewState value=

It is challenging to verify that scripts really work as expected. The script may not work properly, but no errors would be reported. Considering the difficulties, it is suggested to use several ways to verify tests together.

Check the System State

In our case, each user keeps the last state: what view and schedule is opened. If the script switches between views or opens schedules or templates, it is possible to login manually and verify that the system state was really changed.

Some actions, like submit tasks, change the state of the task (from open to closed). It may be verified after the test. If something was created / edited, it also may be verified manually.

Logs

Compare the logs for a script run with the log for the manual execution of the same steps. If there is any difference, it may be a concern. There are some warnings / errors even for the manual execution of the same steps, so it is important to compare both logs.

Text Checks

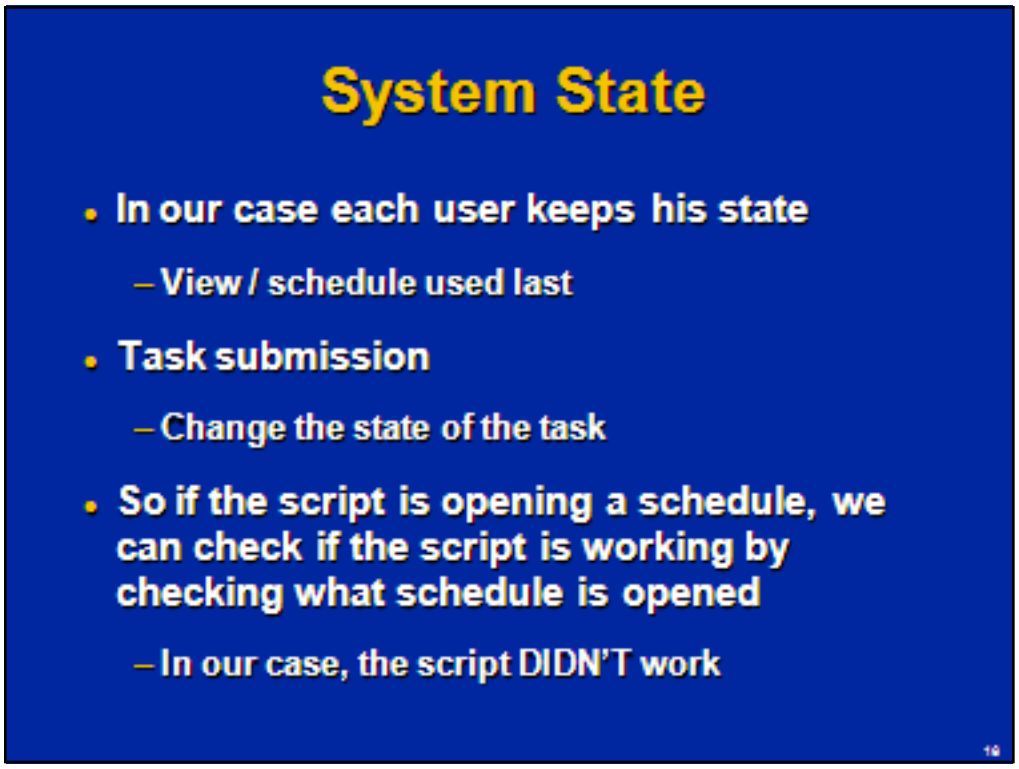
Usually checking for some text (with web_reg_find in LoadRunner) is a good way to verify getting of expected result from the server. There is a setting in recording options that even create such checks automatically – however almost no such checks were generated for the product. Looks like this approach may be not very effective for the product (as far as it uses AJAX and returns only subsets of information).

Global Verification

The following global verification statement allows catching many errors and is very effective for the product:

web_global_verification("Text=error status", "Fail=Found","ID=errorStatus", LAST);

The other global verification statement may catch some other errors.

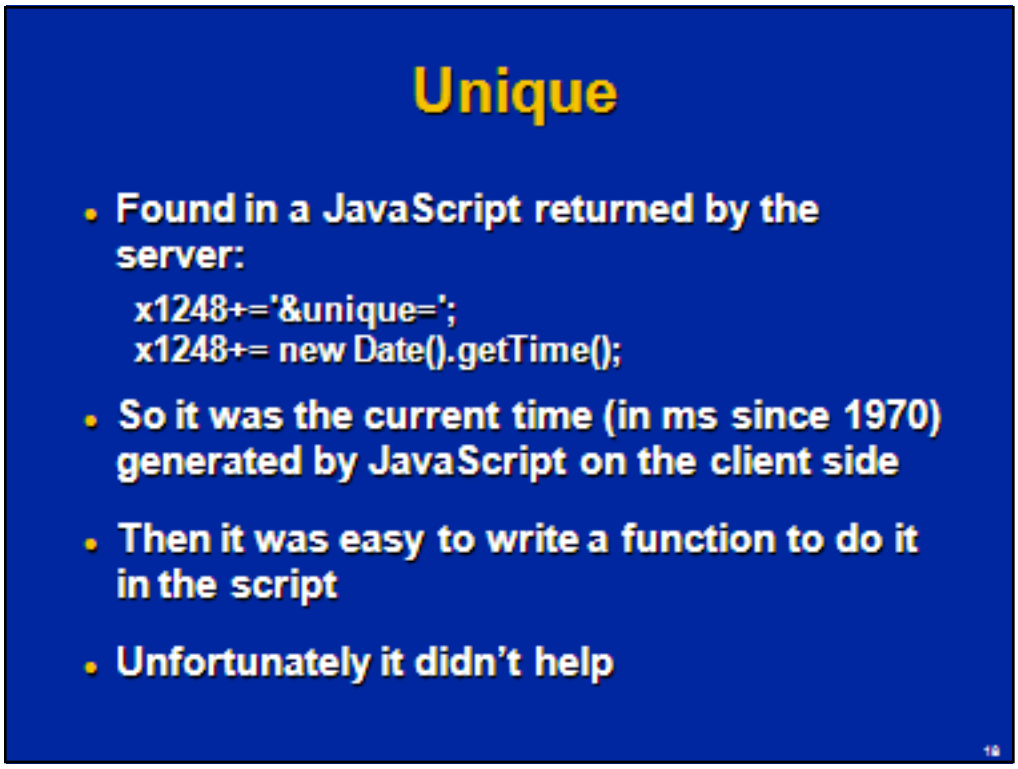


In our case each user keeps his state: view / schedule used last.

There were no LoadRunner errors, nor clear errors in the product logs (although there were few difficult to interpret warnings in addition to warnings generated during manual work). Still the script didn't change the state of the system. For example, the script was supposed to open schedule2. User1 has schedule1 opened. After the script was run, we log in as user1 - and still see schedule1 (if the script worked, it should be schedule2). So the script didn't work and it should be something else there to correlate.

Trying to figure out why scripts don't work (before it was found that headers are necessary) we paid attention to "unique" field in some HTTP requests which was not coming from the server (so can't be correlated):

```
web_url("FCCDashboard.jspx_2",
"URL=http://vulture1:19000/fcc/faces/oracle/apps/epm/fcc/ui/page/FCCDashboard.jspx?_adf.ctrl-state={par_CtrlState25}&Adf-Rich-Message=true&unique=1273258068265&oracle.adf.view.rich.STREAM=rgnbi:0:CNP0:TABLE&javax.faces.ViewState={par_ViewState35}&oracle.adf.view.rich.forceHTML=true",
"Mode=HTML",
LAST);
```



We found it in a JavaScript returned by the server:
x1248+='&unique=';
x1248+= new Date().getTime();

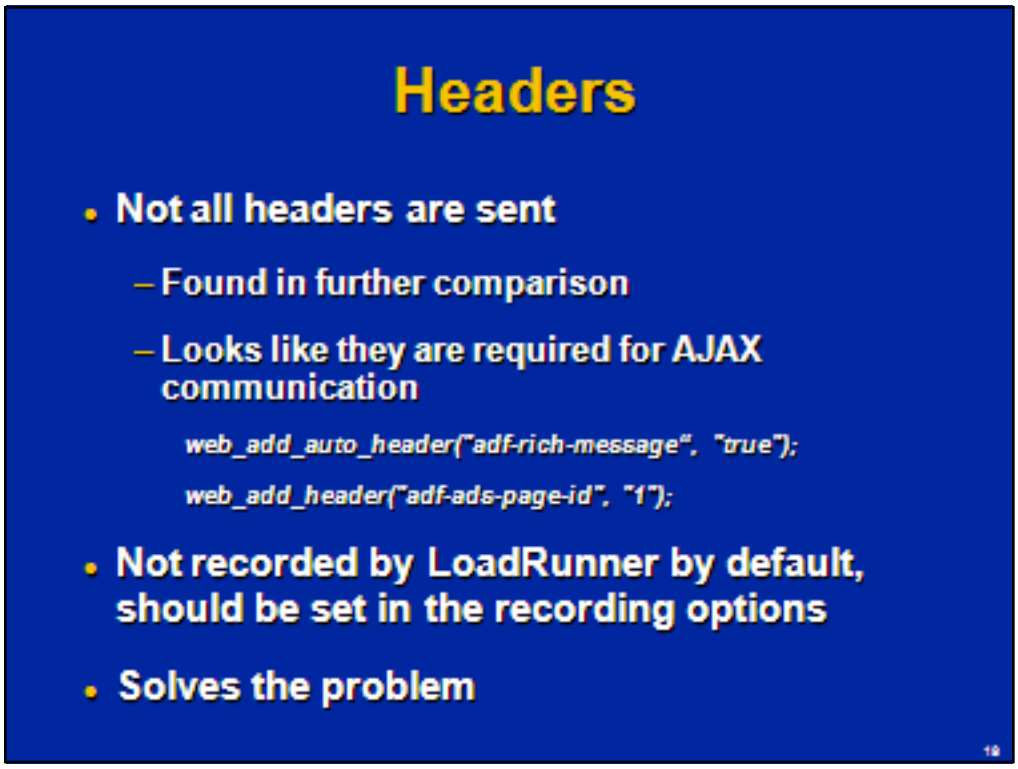
So it is the current time (in ms since 1970), generated by JavaScript on the client side. Then it was easy to write a function to do it in the script:

```
typedef long time_t;
   struct _timeb {
      time_t time;
      unsigned short millitm;
      short timezone;
      short dstflag;
   };

struct _timeb t;

void uTime(){
                        ftime( &t );
                        lr_param_sprintf("par_Time", "%ld%u", t.time, t.millitm);}

_tzset(); // Sets variables used by ftime
```

And the call uTime() before each request with unique to get the current system time:

```
                        uTime();
                        web_url("FCCDashboard.jspx_2",
"URL=http://hfmweb:19000/fcc/faces/oracle/apps/epm/fcc/ui/page/FCCDashboard.jspx?_adf.ctrl-state={par_CtrlState22}&Adf-Rich-Message=true&unique={par_Time}&oracle.adf.view.rich.STREAM=rgntm:0:panReg:0:hv1,rgntm:0:ttReg:0:tTypesL&javax.faces.ViewState={par_ViewState23}&oracle.adf.view.rich.forceHTML=true",
```

However no difference was found in the behavior with and without parameterization of "unique" (the problem was headers, not "unique"), so it left the question if it is needed opened. No feedback from development.



In further comparison of manual execution (the LoadRunner recording log) and script playback it was found that not all headers are sent during playback. It turned out that some headers are important and the script doesn't work without them. These required headers are not recorded by LoadRunner by default, LoadRunner recording options should be modified to record them.

web_add_header("ora_epm_ctg", "{token_14_URL}"); //required for authentication

//required to work properly, although no errors without them

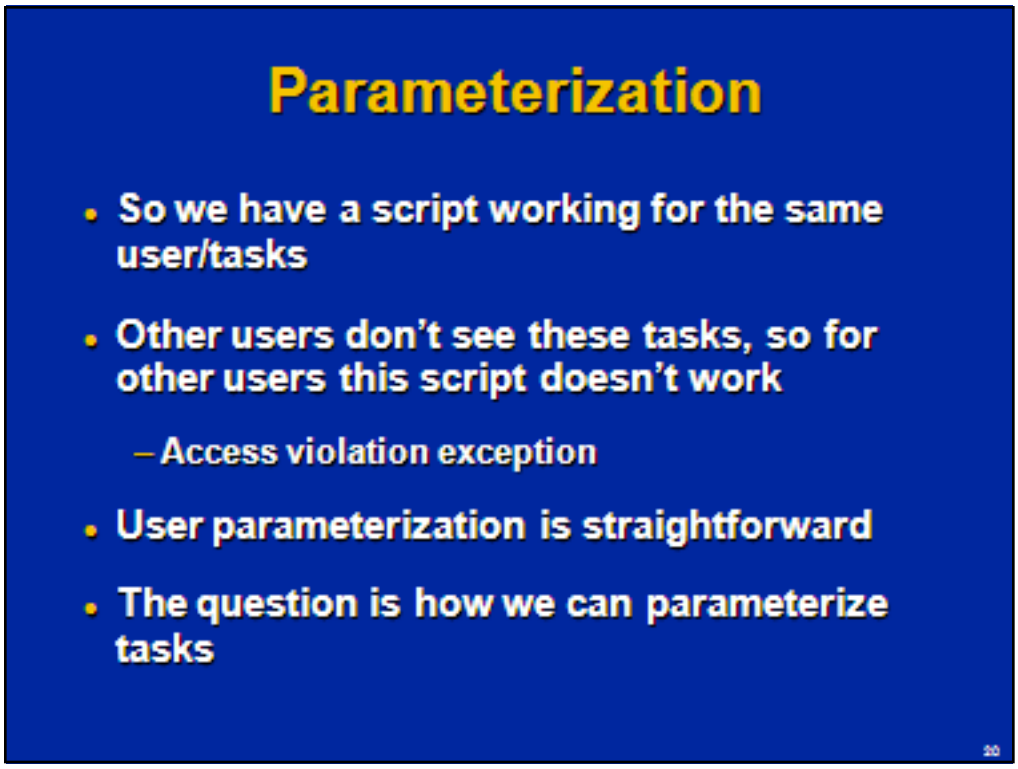
web_add_auto_header("Cache-Control","no-cache");

web_add_auto_header("adf-rich-message", "true");

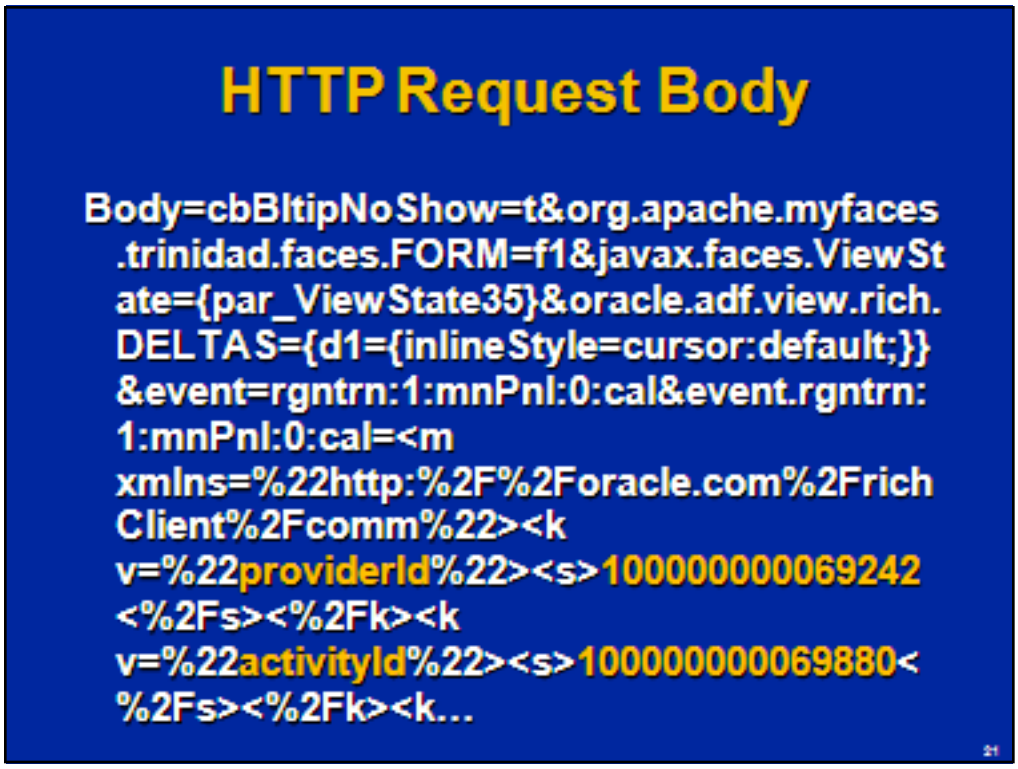
web_add_header("adf-ads-page-id", "1");

It looks like a LoadRunner-specific feature. At least some other products don't discriminate any part of the header and record everything as is (for example, Oracle Load Testing - part of Oracle Application Testing Suite, former Empirix).

Recording these headers solves the problem: the script started to work as expected, changing the status of the systems as requested.

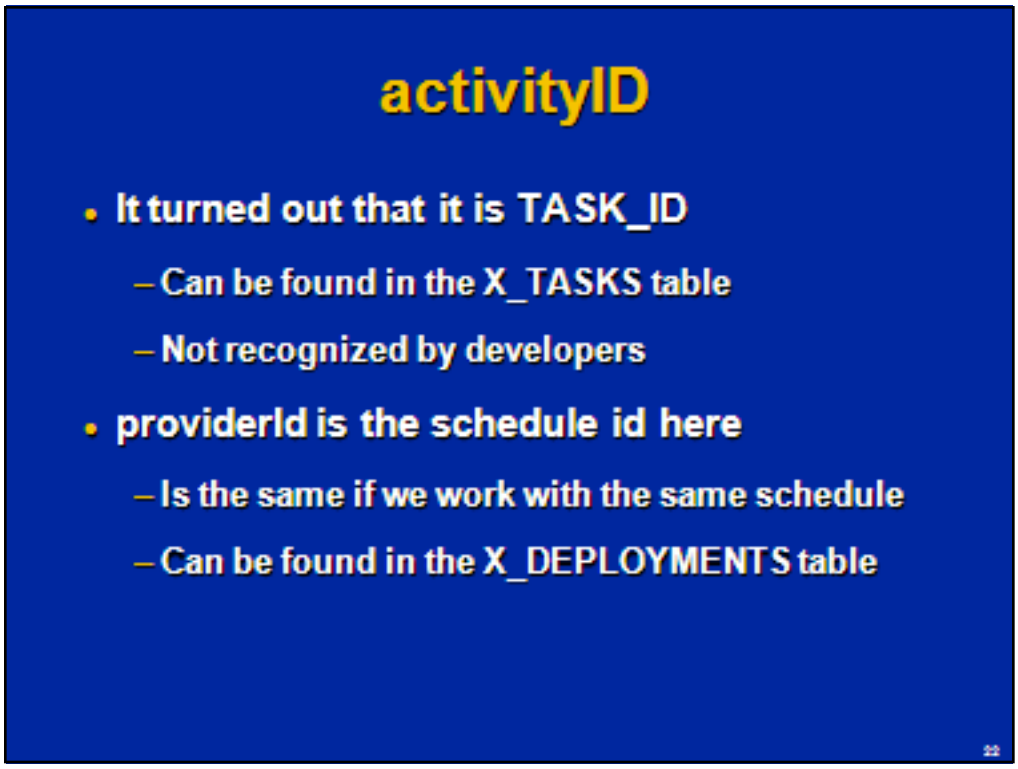


So we have a script working for the same user/tasks. Other users don't see these tasks, so for other users this script doesn't work (when they try to access tasks they don't have access to, a access violation exception is generated). User parameterization is straightforward (just parameterize user's name). The question is how we can parameterize tasks.



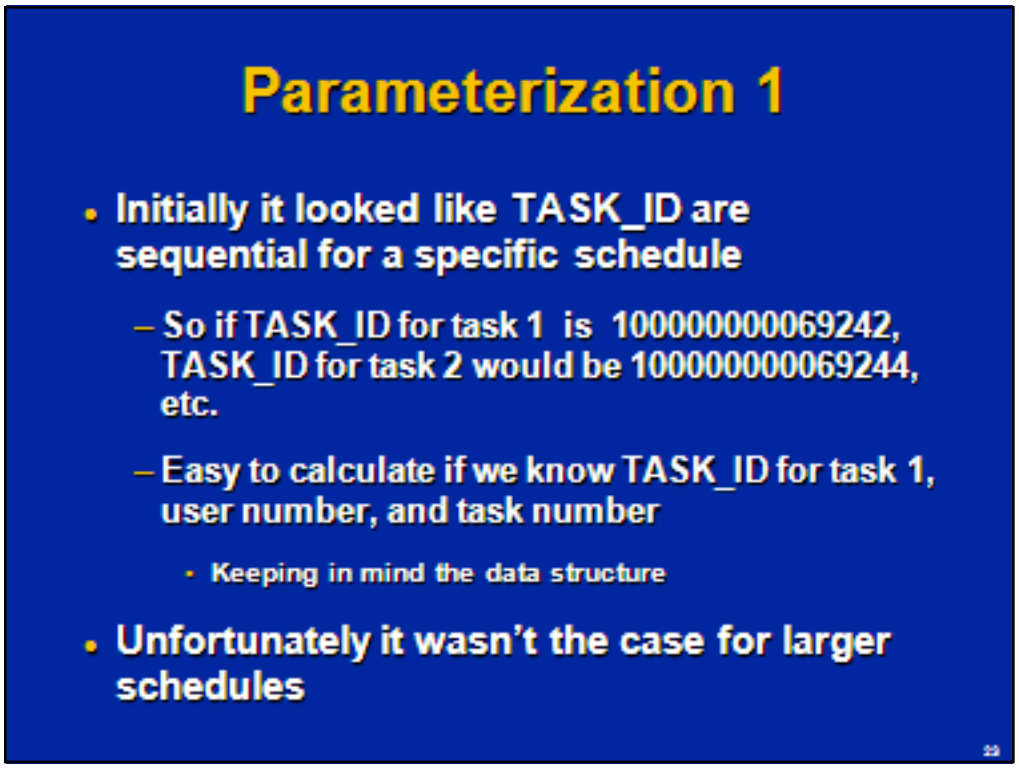
Analyzing HTTP request related to opening tasks we found two parameters that looked like they may be related to a specific task: providerId and activityId. If we opened another task, activityId was different.

```
web_custom_request("FCCDashboard.jspx_20",
                   "URL=http://vulture1:19000/fcc/faces/oracle/apps/epm/X/ui/page/XDashboard.jspx?_adf.ctrl-state={par_CtrlState25}",
                                   "Method=POST",
                   "Body=cbBItipNoShow=t&org.apache.myfaces.trinidad.faces.FORM=f1&javax.faces.ViewState={par_ViewState35}&oracle.adf.view.rich.DELTAS={d1={inlineStyle=cursor:default;}}&event=rgntrn:1:mnPnl:0:cal&event.rgntrn:1:mnPnl:0:cal=<m
xmlns=%22http:%2F%2Foracle.com%2FrichClient%2Fcomm%22><k
v=%22providerId%22><s>100000000069242<%2Fs><%2Fk><k
v=%22activityId%22><s>100000000069880<%2Fs><%2Fk><k
v=%22keyStroke%22%2F><k
v=%22clickCount%22><n>1<%2Fn><%2Fk><k
v=%22button%22><s>LEFT<%2Fs><%2Fk><k v=%22triggerType%22"
                                   "><s>MOUSE<%2Fs><%2Fk><k
v=%22type%22><s>calendarActivity<%2Fs><%2Fk><%2Fm>&oracle.adf.view.rich.PROCESS=rgntrn:1:mnPnl:0:cal",
                                   LAST);
```



It turned out that activityId is TASK_ID and can be found in the X_TASKS table. It was not recognized by developers (due to using of a different name?) and was found by blind looking in whatever places it could be in – luckily the table structure and naming were pretty simple and we had full access to the database.

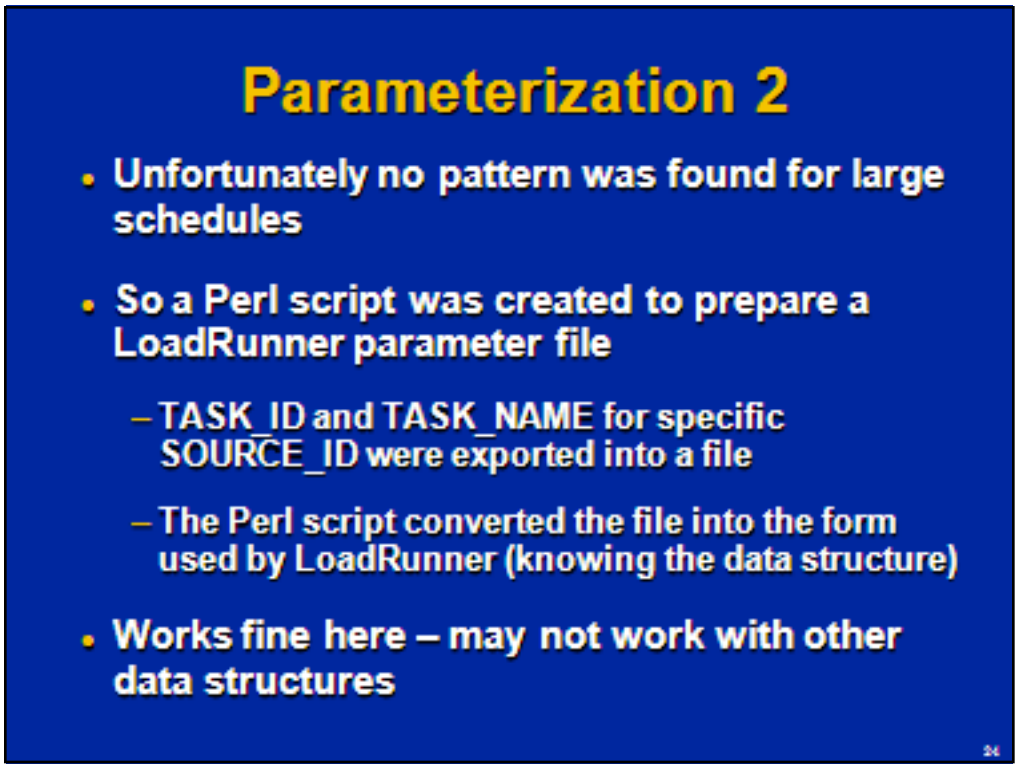
providerId is the schedule id here and is the same if we work with the same schedule. Can be found as DEPLOYMENT_ID in the X_DEPLOYMENTS table.



Initially it looked like TASK_ID are sequential for a specific schedule. So if TASK_ID for task 1 is 100000000069242, TASK_ID for task 2 would be 100000000069244, etc. Then it is easy to calculate if we know TASK_ID for task 1, user number, and task number (keeping in mind the data structure):

```
uNum = atoi(lr_eval_string("{par_userNum}"));
tNum = atol(lr_eval_string("{par_task1}")) + (uNum-1)*2 + (atoi(lr_eval_string("{par_taskNum}"))-1)*40;
```

Unfortunately it wasn't the case for larger schedules



Unfortunately no pattern was found for large schedules. So a Perl script was created to prepare a LoadRunner parameter file.

First, ids were extracted from the database. In the X_TASKS table there are fields TASK_ID (activityId in the request) and SOURCE_ID (providerId in the request). So TASK_NAME and TASK_ID may be selected for the specific SOURCE_ID (which may be found as DEPLOYMENT_ID in the X_DEPLOYMENTS table using DEPLOYMENT_NAME). So TASK_ID and TASK_NAME for specific SOURCE_ID were exported into a file.

The Perl script converted the file into the form used by LoadRunner (knowing the data structure):

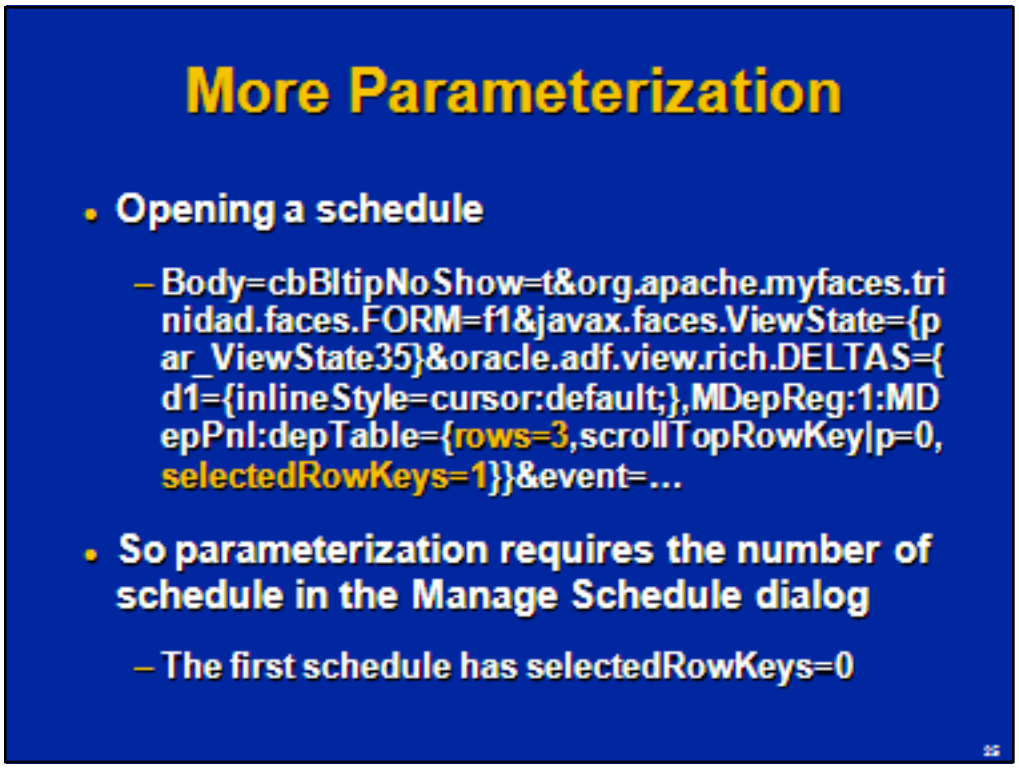
userNum, Task1, Task2, Task3, Task4, Task5

user0001,100000000578740,100000000578900,100000000578650,100000000578286,100000000578448

user0002,100000000578742,100000000578902,100000000578608,100000000578288,100000000578450

…

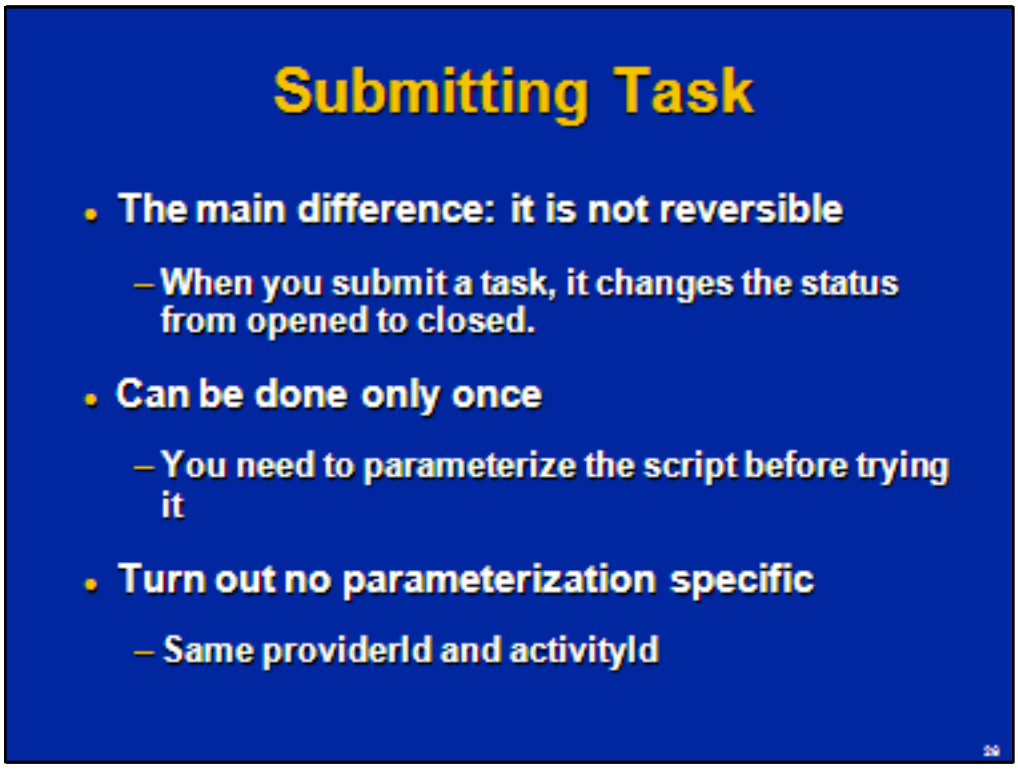
Works fine here – may not work with other data structures.



Parametrization would be different for other actions. For example, for opening a schedule we see the following in the HTTP request body:

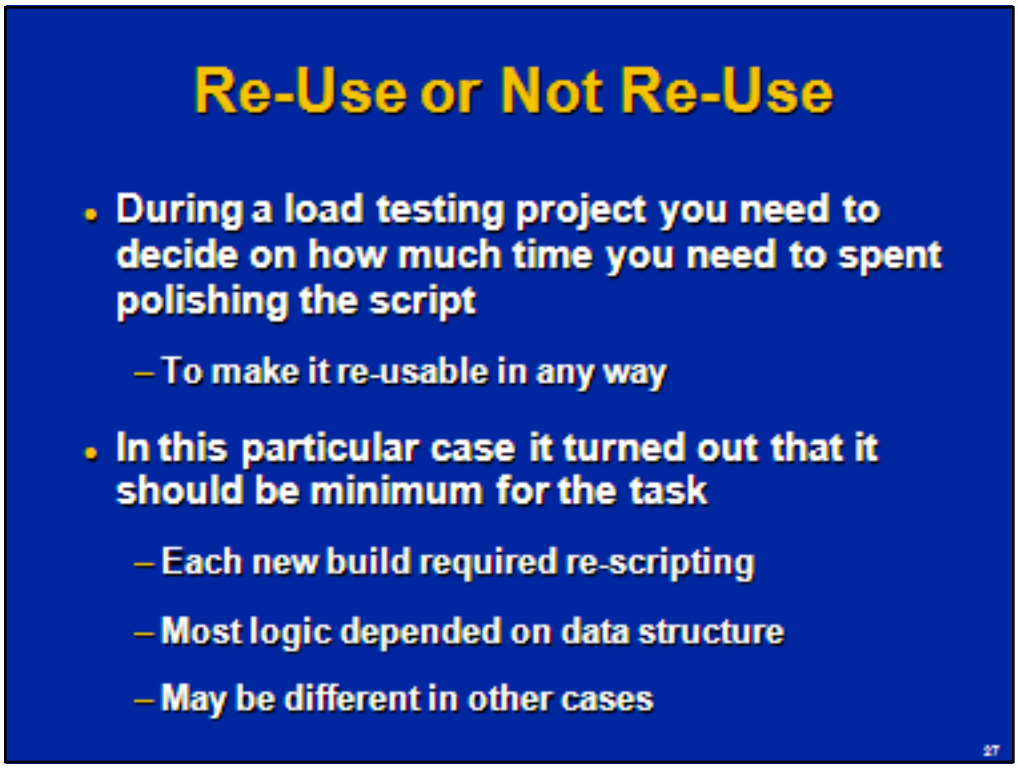
Body=cbBltipNoShow=t&org.apache.myfaces.trinidad.faces.FORM=f1&javax.faces.ViewState={par_ViewState35}&oracle.adf.view.rich.DELTAS={d1={inlineStyle=cursor:default;},MDepReg:1:MDepPnl:depTable={**rows=3**,scrollTopRowKey|p=0,**selectedRowKeys=1**}}&event=…

So parameterization requires the number of the schedule in the Manage Schedule dialog. The first schedule has selectedRowKeys=0, the second 1, the third 2, etc.



The main difference for "submit" task is that it is not reversible. When you submit a task, it change the status from opened to closed.

So it can be done only once for a specific task. You need to parameterize the script before trying it. And it involves another component: SOA Suite. So it was considered as a separate scripting test. However, it turned out that there was no parameterization specific there. Same providerId and activityId.

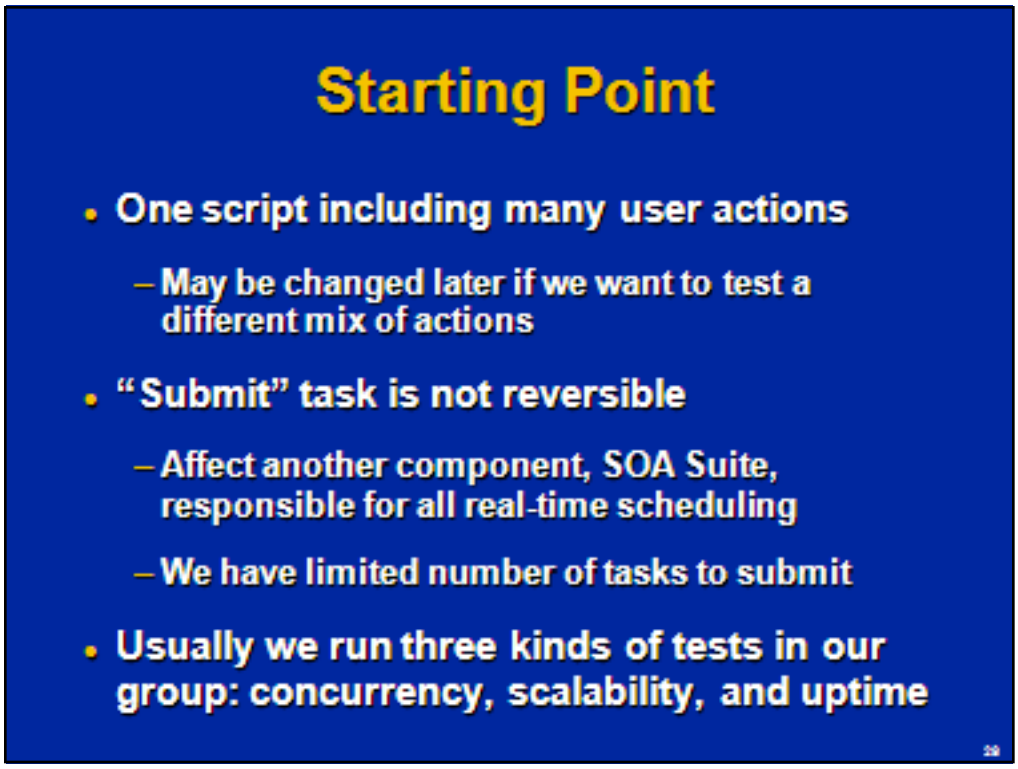


Another interesting question: during a load testing project you need to decide on how much time you need to spent polishing the script. Do you need to make it re-usable in anyway?

In this particular case it turned out that it should be minimum of polishing for the current task on hands. Each new build required re-scripting, most logic depended on data structure. So trying to make it re-usable / making more generic logic will be wasting of very limited time here. It may be different in other cases

Agenda

- System
- Scripting
- Testing



In most tests we used one script including many user actions (like login, open the application, switch views, opening a schedule, opening a task, submitting a task, refresh, show dashboards, filtering information, etc.). It was mainly dictated by efficiency (need to re-script for each build, about the same amount of efforts required for parameterization of a small script as for parameterization of a large script). It may be changed later if we want to test a different mix of actions. In some cases a auxiliary script was created to eliminate irregularities (such as first usage pop-up messages).

A serious issue is that "Submit" task is not reversible, but is a very important piece of functionality affecting another component, SOA Suite, responsible for all real-time scheduling. "Open" task doesn't touch SOA Suite. And we have limited number of task to submit.

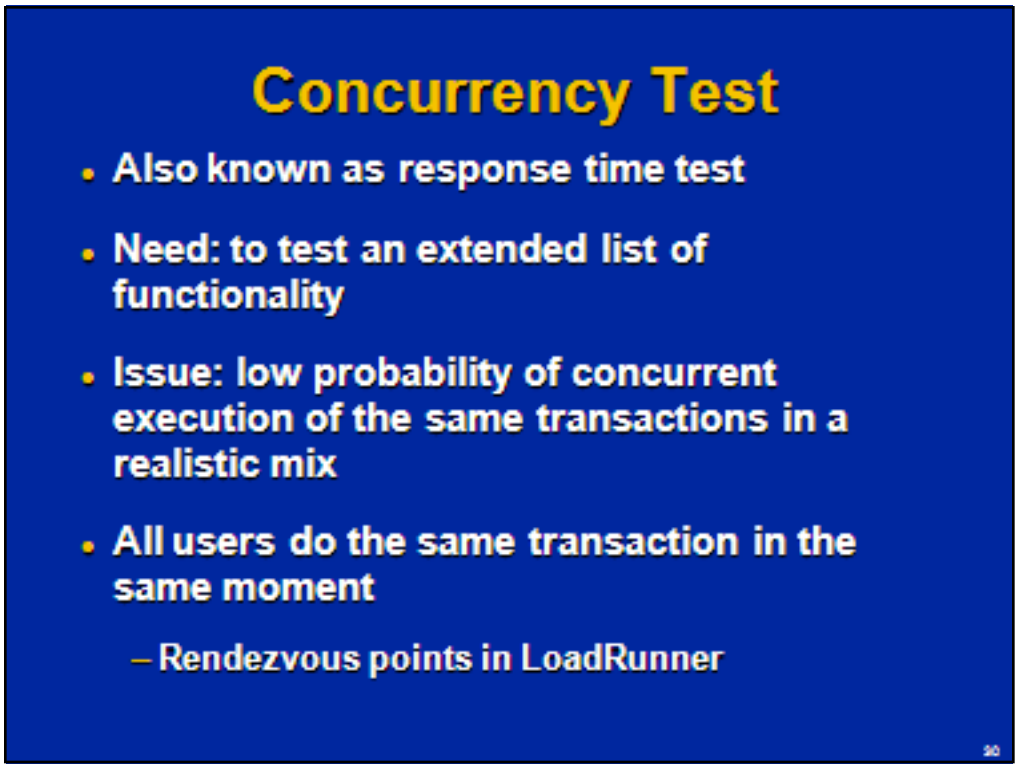
Usually we run three kinds of tests in our group:

-Response times (concurrency) tests

-Scalability tests

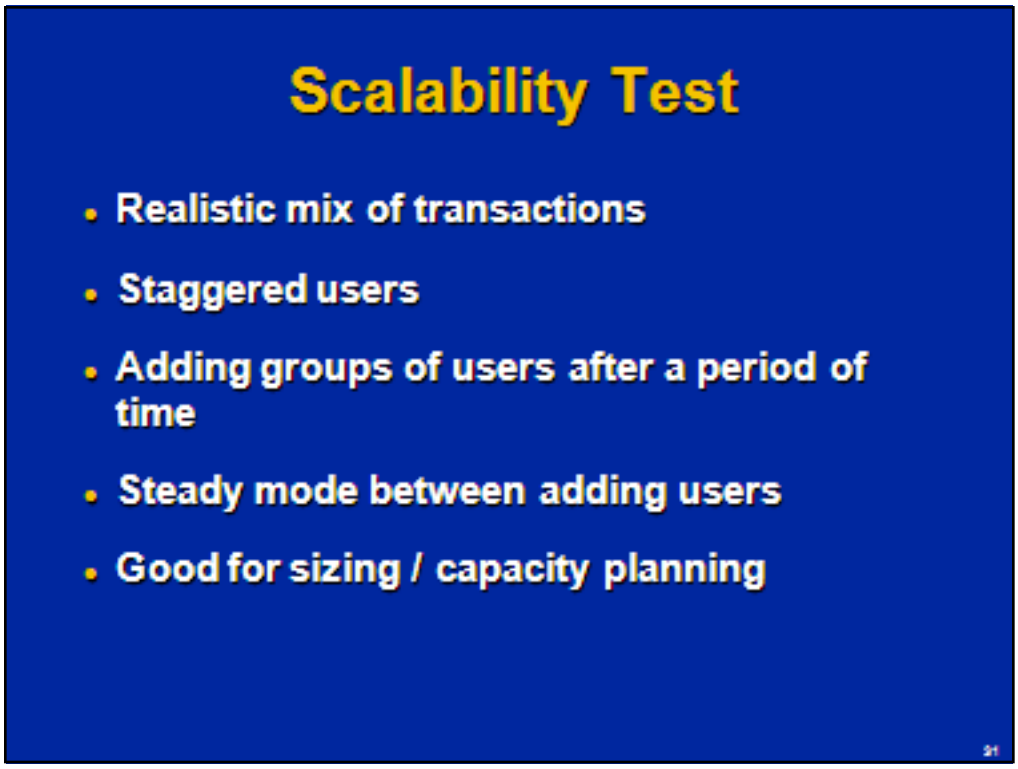
-Uptime tests.

This was the way it was when I came to the group, so I am not sure who should be credited for the approach. But the more I think about it, the more sense it makes (at least for a group responsible for testing new software builds required to provide maximum information for the limited amount of time and working closely with development to fix the issues).

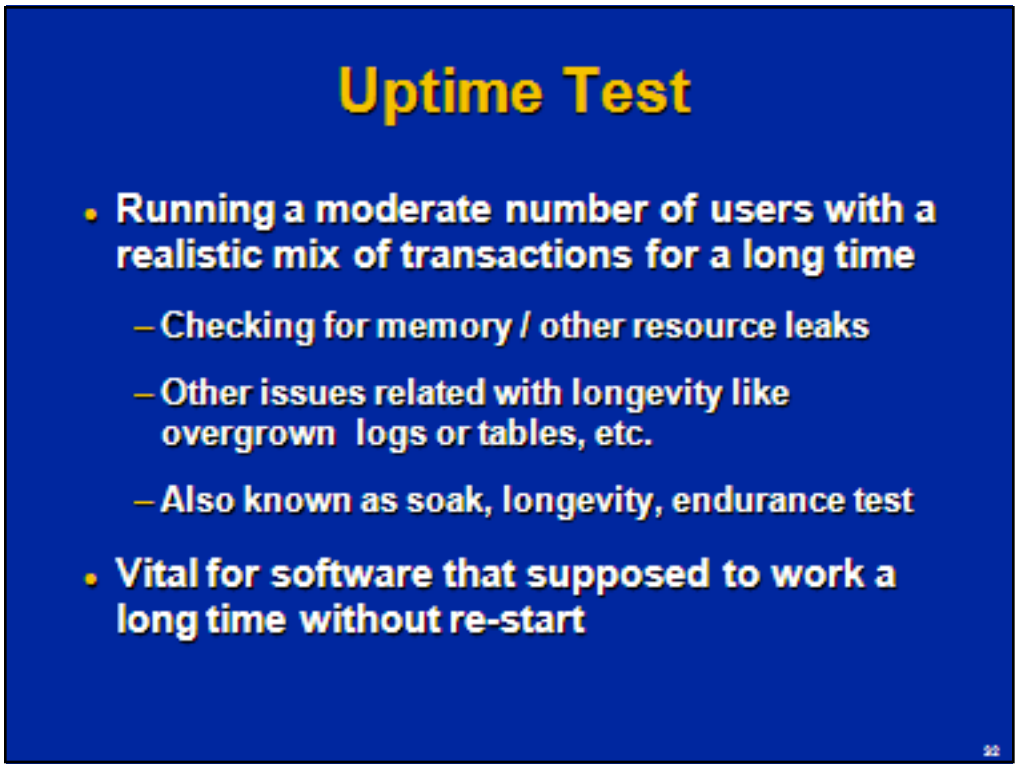


The concurrency test is designed to test an extended list of functionality and overcome a known issue: low probability of concurrent execution of the same transactions in a realistic mix when you have a lot of different short transactions. To address this issue the concurrency test (also known as response time test) was introduced: when all users do the same transaction in the same moment (rendezvous points in LoadRunner).

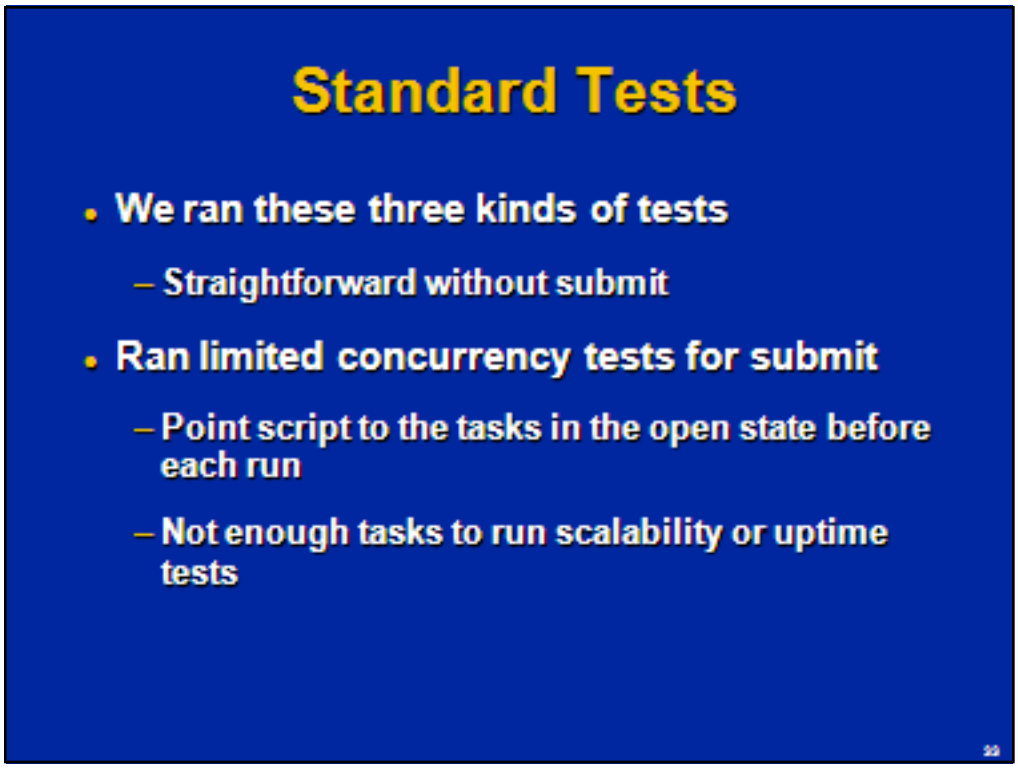
I haven't seen mentioning of this test in this context, but it make perfect sense and address one of very serious issues in load testing: limited functionality coverage. Concurrency tests are not realistic, but allow to test a lot of functionality and find if any specific transaction doesn't scale or doesn't behave.



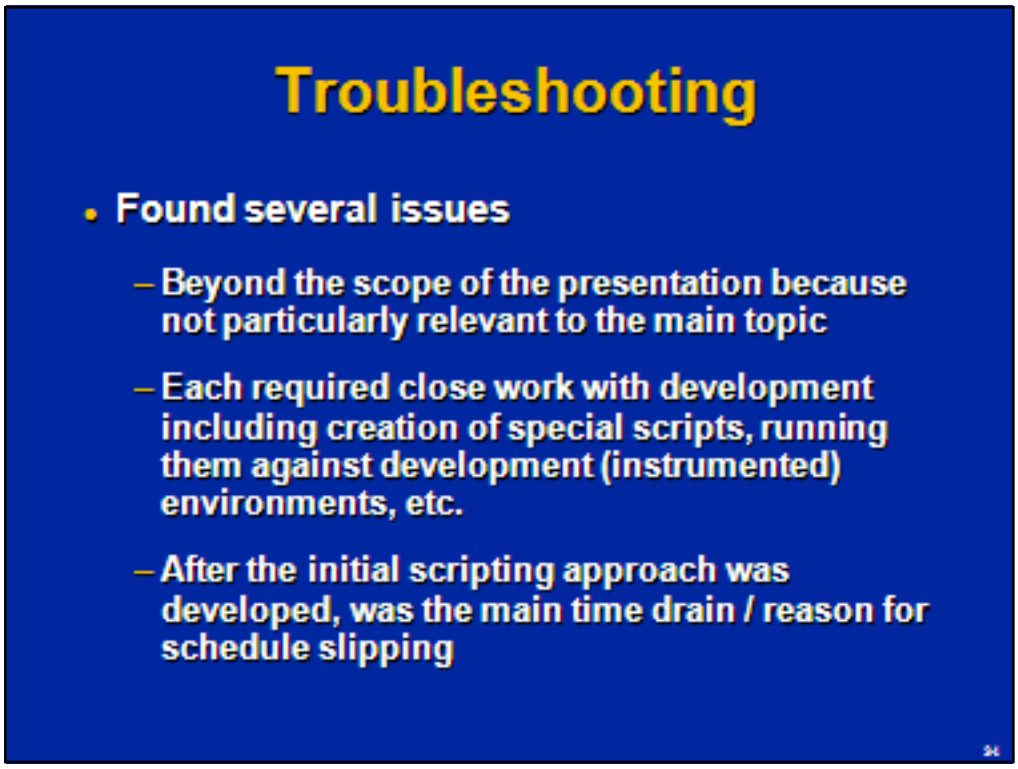
Scalability test is a “realistic” test for different levels of load. It includes a realistic mix of transactions, staggered users, and adding groups of users after a period of time. The main condition is that the test reach the steady state on each level of load – otherwise results would be misleading. Combining several levels of loads in a single test speed up the testing and may make trends more visible, but it also may make analysis and troubleshooting more complex. The results of the scalability test are good for sizing and capacity planning. For example, a group of 50 users may be added each 30 minutes while response times are below 20 seconds.



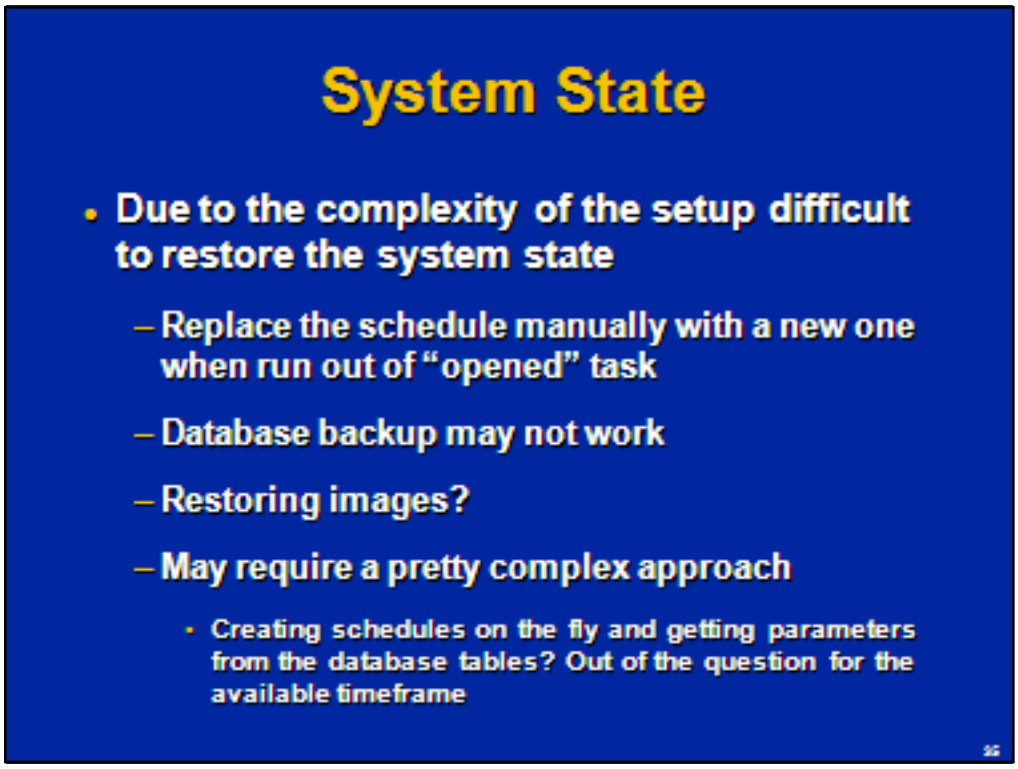
Uptime test includes running a moderate number of users with a realistic mix of transactions for a long time (usually from several hours to several weeks depending on the situation). Also known as soak, longevity, or endurance test. It is checking for memory / other resource leaks as well as other issues related with longevity like overgrown logs or tables, etc. It is vital for software that supposed to work a long time without re-start.



We ran these three kinds of tests, which were straightforward without submitting tasks. We ran limited concurrency tests for submit, pointing the script to the tasks in the open state before each run. We don't have enough tasks to run scalability or uptime tests, which was a clear limitation of the tests run.

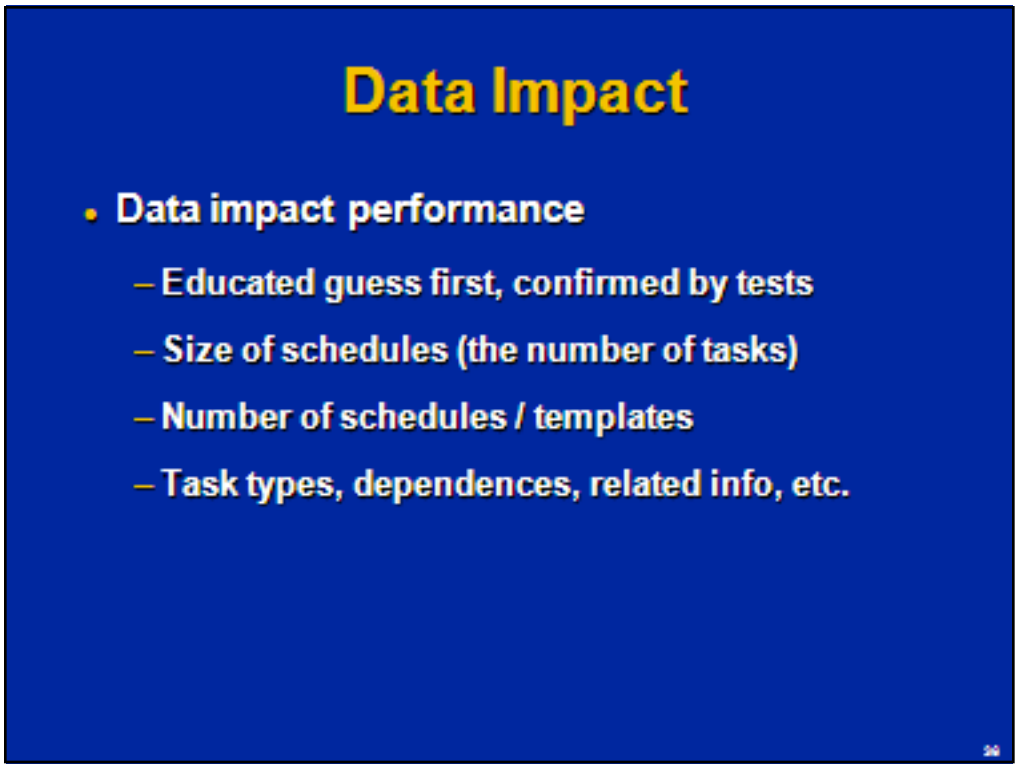


Several issues were found during the testing. They are beyond the scope of the presentation because not particularly relevant to the main topic. However it is worth time to mention that each issue required close work with development including creation of special scripts, running them against development (instrumented) environments, etc. After initial scripting approach was developed, the issue were the main time drains / reasons for schedule slipping. Still it is the very reason for performance testing and the main its achievement is that the final product was shipped without these issues, not the number of tests ran or adherence to the original schedule.

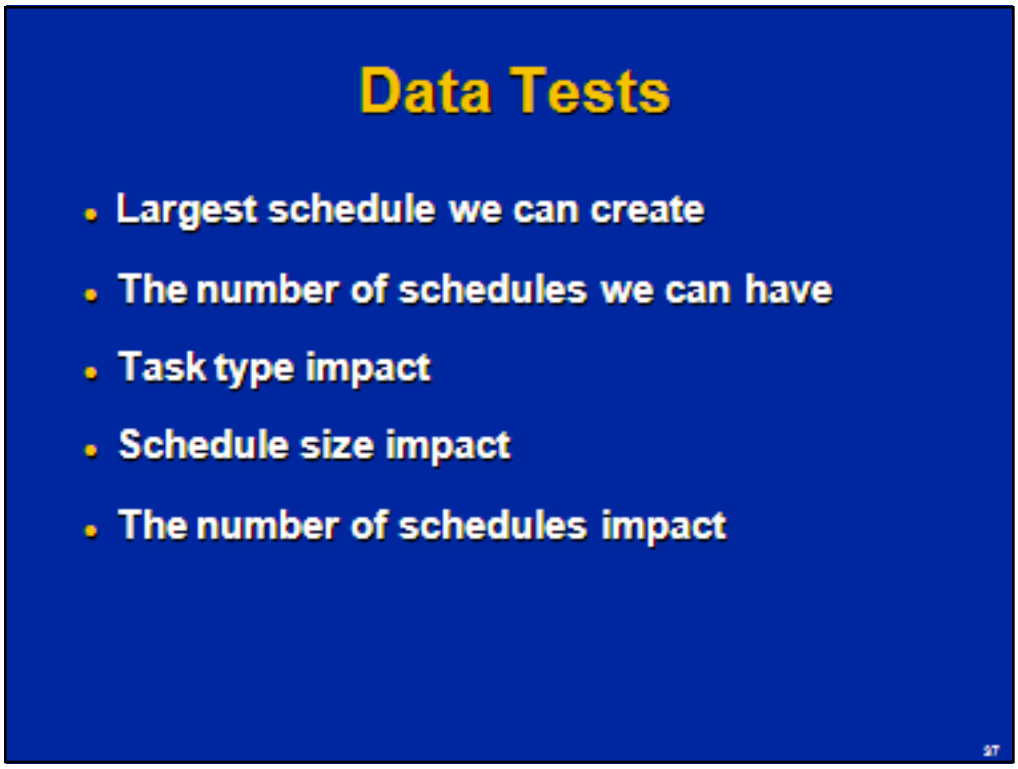


Due to the complexity of the setup it is difficult to restore the system state. During the tests, we replaced the schedule manually with a new one when run out of "opened" task.

It is not a completely clean approach and requires to re-create the parameter file. The ideal case would be to restore the system back to exactly the same state. However it is very difficult due to complexity of the setup. Database backup may not work. Maybe creating disk images and restoring them may work. It may require a pretty complex approach, like creating schedules on the fly and getting parameters from the database tables. Unfortunately it was out of the question for the available timeframe.

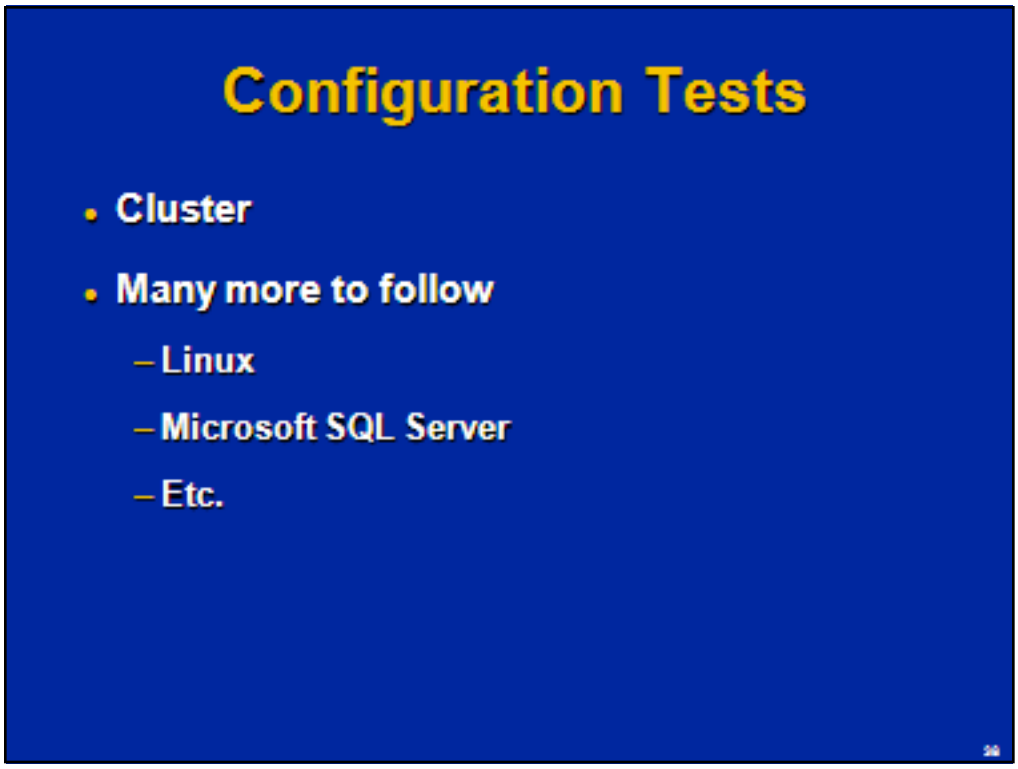


Data definitely impact the system performance. It was an educated guess first, then confirmed by tests. The most important data metrics were the size of schedules (the number of tasks) and the number of schedules / templates. Other data metrics (like task types, dependences, related info) may impact performance too.

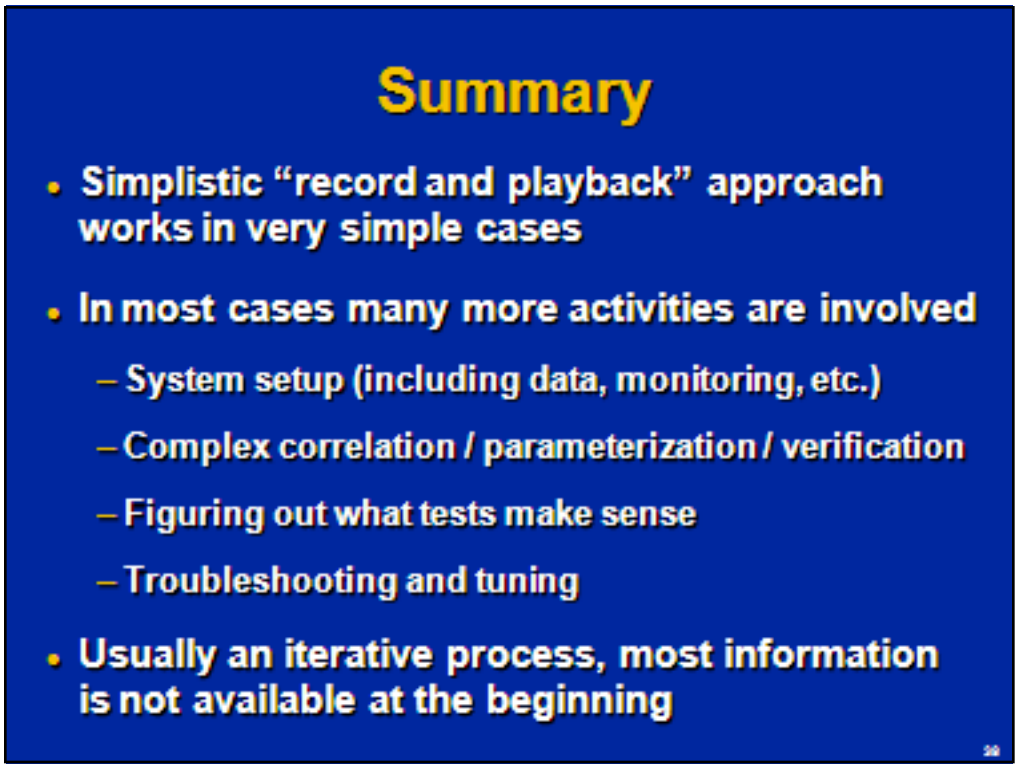


Multiple tests were run to find out data impact, many of them were single-user manual tests. The largest schedule we can create.

The number of schedules we can have. As well as standard LoadRunner tests with different task types, different schedule size, different number of schedules to figure out their impact on response times and resource utilization.



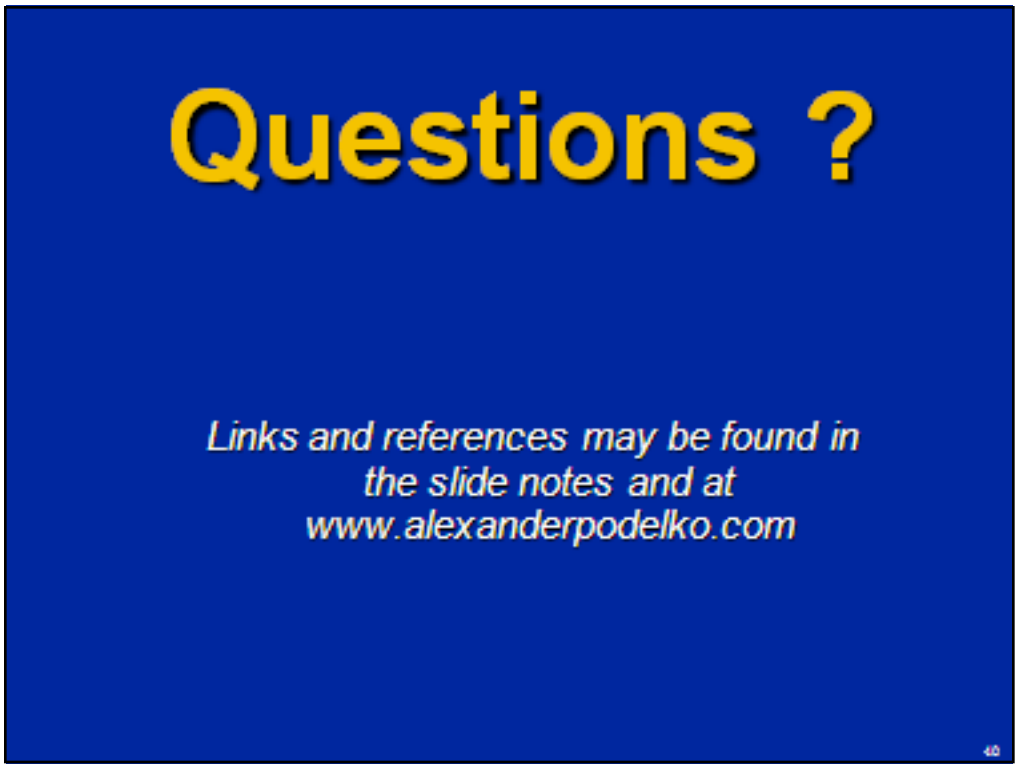
Only configuration test ran was for a cluster configuration (two application instances on the different servers). Many more configuration tests are planned, including using Linux instead of Windows, Microsoft SQL Server instead of Oracle Database, etc.



The simple "record and playback" approach works in very simple cases. In most cases many more activities are involved, including (but not limited):

- System setup (including data generation, monitoring, etc.)
- Complex correlation / parameterization / verification
- Figuring out what tests make sense
- Troubleshooting and tuning (mainly left outside of this presentation).

Load testing is usually an iterative process, where most information is not available at the beginning.



References

[LoadRunner] HP LoadRunner documentation.

[Molyneaux09] Molyneaux I. The Art of Application Performance Testing. O'Reilly, 2009.

[Performance07] Meier J.D. , Farre C., Bansode P., Barber S., Rea D. Performance Testing Guidance for Web Applications, 2007

http://perftestingguide.codeplex.com/

[Podelko08] Podelko A. Agile Performance Testing, CMG, 2008.

http://www.alexanderpodelko.com/docs/Agile_Performance_Testing_CMG08.pdf

[Podelko06] Podelko A. Load Testing: Points to Ponder, CMG, 2006.

http://www.alexanderpodelko.com/docs/Points_to_Ponder_CMG06.pdf